Diplomarbeit

A unit-test platform for design tools for fault-tolerant real-time systems

Ausgeführt am Institut für

Technische Informatik

der Technischen Universität Wien

unter der Anleitung von

Ao. Univ.-Prof. Dr. Peter Puschner

und

Dipl.-Ing. Dr. Raimund Kirner

als verantwortlich mitwirkendem Universitätsassistenten

Branislav Križan
Matr.-Nr. 0025634
Ormisova 3, 831 02 Bratislava, Slowakei

Wien, März 2007

Acknowledgments

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. I want to thank to TTTech Company, especially to Mr. Dipl. Ing. Alois Goller to allow me to write my thesis at theirs and to encourage me in it.
I am deeply indebted to my supervisor Univ. Ass. PhD Raimund Kirner from the Department of Technical Informatics from Viennna University of Technology whose help, stimulating suggestions and encouragement helped me in all the time of research for writing this thesis.
My colleagues from the TTTech Company supported me in my research work. I want to thank them for all their help, support, interest and valuable hints. Especially I am obliged to Software Tool Team, Christoph Zuschnig and Karl Salasch.
I thank to my father who offered me suggestions for improvement and to my mother who helped me with the English style and grammar.

Abstract

Due to the increasing complexity of software, software reliability nowadays plays a very important role. Software reliability can be increased by improving the software development process and the quality of software tests. TTTech develops software to design dependable real-time systems, thus keeping the reliability of the developed software at a high level. This master's thesis focuses on the assessment of software tests used for checking the reliability of software developed on Python platform. In particular, this master's thesis is aimed at examining the testing methods, code metrics and the implementation of a new unit test framework containing new additional metrics. The new unit test framework, which also replaces the current unit test framework, should provide additional metrics to assist developers in writing better-quality software tests and source codes. The results of the new unit test framework can be used as input data for quality management as well.

Zusammenfassung

Auf Grund der zunehmenden Komplexität von Software spielt die Funktionsfähigkeit von Software eine außerordentlich wichtige Rolle. Die Funktionsfähigkeit von Software kann durch eine Verbesserung des Entwicklungsprozesses und der Qualität von Software-Tests erhöht werden. Die Firma TTTech entwickelt Software-Produkte für den Entwurf von zuverlässigen Echtzeitsystemen, wodurch die Funktionsfähigkeit der entwickelten Software auf einem hohen Niveau bleibt. Diese Diplomarbeit beschreibt die Bewertung von Software-Tests, welche die Funktionsfähigkeit der in Python Plattform entwickelten Software prüfen, und untersucht insbesondere Testmethoden, Code-Metriken sowie die Implementierung eines neuen Unit-Test-Frameworks, das zusätzliche Metriken enthält. Das neue Unit-Test-Framework, welches das derzeit in Verwendung befindliche ersetzen soll, stellt zusätzliche Metriken zur Verfügung, welche den Entwicklern bei der Entwicklung von qualitativ besseren Software-Tests und Source-Codes behilflich sind. Die Ergebnisse des neuen Unit-Test-Framework können auch als Input für das Qualitätsmanagement verwendet werden.

Contents

Acknowledgement ii

Abstract iii

1 Introduction 1
- 1.1 Definition of Test Case 1
- 1.2 Scope 1
- 1.3 Typographic Conventions 2

2 Current Software Testing Framework 3
- 2.1 The Unit Test Framework 3
- 2.2 The Mechanism of the Unit Test Framework 5
- 2.3 The Analysis of the Unit Test Framework 6
- 2.4 The Unit Test Mechanism 7
- 2.5 The Analysis of the Unit Test 9
- 2.6 The Docstring Test Mechanism 9
- 2.7 The Analysis of the Docstring Test 10
- 2.8 Summary 11

3 Overview of Testing Methods 13
- 3.1 Criteria 14
  - 3.1.1 The Input Criterion 14
  - 3.1.2 The Apriori Knowledge Criterion 14
  - 3.1.3 The Test Control Criterion 15
  - 3.1.4 The Test Nature Criterion 16
  - 3.1.5 The Test Scope Criterion 17
  - 3.1.6 The Tested Items Criterion 17
- 3.2 A List of Known Testing Methods 18
  - 3.2.1 Random Test 18
  - 3.2.2 Regression Test 18
  - 3.2.3 Functional Test 19
  - 3.2.4 Structural Test 19
  - 3.2.5 Symbolic Execution 19
  - 3.2.6 Program Spectra Analysis 20
  - 3.2.7 Anomaly Detection 21
  - 3.2.8 Walk-Through Verification 21
  - 3.2.9 Mathematical Verification 21
  - 3.2.10 Executable Assertions 22
  - 3.2.11 Adaptive Test 22
  - 3.2.12 Object State Testing 23
  - 3.2.13 Unit Test 24
  - 3.2.14 Integration Test 24
- 3.3 Summary 26

4 Research of Code Metrics 27
4.1 Introduction 27
4.2 Control Flow Code Coverage Metrics 27
4.2.1 Statement Coverage Metrics 28
4.2.2 Decision Coverage Metrics 28
4.2.3 Condition Coverage Metrics 28
4.2.4 Decision Condition Coverage Metrics 28
4.2.5 Multiple Condition Coverage Metrics 29
4.2.6 Path Coverage Metrics 29
4.2.7 Modified Condition Decision Coverage Metrics 29
4.2.8 Boundary-Interior Metrics 29
4.2.9 Structured Path Coverage Metrics 29
4.2.10 TER Metrics 30
4.3 Data Flow Code Coverage Metrics 30
4.3.1 All-P-Uses Metrics 31
4.3.2 All-Defs Metrics 31
4.3.3 All-P-Uses Some-C-Uses Metrics 33
4.3.4 All-C-Uses Some-P-Uses Metrics 33
4.3.5 All-Uses Metrics 33
4.3.6 All-DU-Paths Metrics 33
4.3.7 2-DR Interaction Metrics 33
4.3.8 Elementary Data Context Metrics 33
4.3.9 Ordered Data Context Metrics 34
4.3.10 Required Pairs Metrics 34
4.3.11 Required k-tuples Metrics 34
4.4 Code Complexity Metrics 34
4.4.1 Weighted Methods Per Class 35
4.4.2 Depth Of Inheritance Tree 35
4.4.3 Number Of Children 36
4.4.4 Coupling Between Object Classes 36
4.4.5 Response For A Class 36
4.4.6 Lack Of Cohesion In Methods 36
4.4.7 Code Size 37
4.5 Summary 37

5 The New Unit Test Framework 39
5.1 The Requirements and the Limitations Concerning Metrics 39
5.1.1 The Company Requirements for the new Unit Test Framework 39
5.1.2 The Abstract Syntax Tree 39
5.1.3 The Tracking System 40
5.2 The Arguments About the Metrics 42
5.2.1 The Chosen Control Flow Code Coverage Metrics . . 42
5.2.2 The Rejection of The Data Flow Code Coverage Metrics 43
5.2.3 The Problems With The Code Complexity Metrics . . 43
5.3 The Design of the Metrics 44
5.3.1 The Statement Coverage 44
5.3.2 The Decision Coverage 45
5.3.3 The Code Size 47

5.3.4 The Execution Time Per Method 47
5.3.5 The Weighted Method Per Class 47
5.3.6 The Response For A Class 48
5.4 The Input and The Output of The New Unit Test Framework 48
5.4.1 Framework Command Line Input 48
5.4.2 Framework XML Output 49
5.5 The Architecture of the new Unit Test Framework 57
5.5.1 The Tracker Component 60
5.5.2 The Parser Component 61
5.6 Summary 62

6 Evaluation **64**
6.1 Metrics Evaluation 69
6.2 Internal Data Structures Evaluation 83
6.3 Time Measurement Precision 95
6.4 Summary 96

7 Conclusion **97**

Glossary **99**

References **100**

Index **102**

1 Introduction

TTTech offers a range of software tools that enable developers of aerospace, automotive, and industrial control equipment to deliver reliable embedded systems quickly and efficiently. One of the most important features of these software tools is dependability with a couple of processes, standards and testing frameworks ensuring the quality of these tools.

This master's thesis deals with a unit-test framework that is part of the testing frameworks used at *TTTech*. This master's thesis aims at analyzing and integrating software metrics based on source code. Those metrics are to provide information on the quality of unit tests executed in the unit-test framework, which results in an improved unit-test framework with new software metrics.

1.1 Definition of Test Case

Generally, a *test case* for any system is an item that specifies the input behavior of that system and the expected output behavior of that system [BJK+05]. We have to distinguish between test case and test case implementation. A test case is only the specification. This master's thesis deals with two types of *test case implementation*s: test case function and test case string pair.

A *test case function* is a function or method simulating the input behavior of a system and implementing the output behavior of that system, with the output behavior being the result of a comparison between the expected behavior and the behavior actually observed during testing.

A *test case string pair* is a pair of command line strings specifying the command line input behavior of a system and the command line output behavior of that system. Test case string pairs are always included as comments in the module to be tested.

The two types of test case implementations described above have identical logical steps: execution, comparison and result delivery.

1.2 Scope

This master's thesis covers the following topics:

- Section 2 on page 3, **Current Software Testing Framework**, analyzing the unit-test framework currently used;
- Section 3 on page 13, **An Overview of the Testing Methods**, giving an overview of various testing methods including the unit test;
- Section 4 on page 27, **Research of Code Metrics**, describing various kinds of metrics used for the evaluation of the source code;

- Section 5 on page 39, **The New Unit Test Framework**, describing the design and the user interface of the new unit-test framework to be used;
- Section 6 on page 64, **Evaluation**, presenting the results of the new unit-test framework used.

1.3 Typographic Conventions

The typographic conventions for this master's thesis are as follows:

Element	**Typographic format**	**Example**
New definitions	italics	This is a *definition*
A software object	typewriter	This is an `object`

2 Current Software Testing Framework

This section deals with a method of software testing currently used at *TTTech*. This method includes unit tests and docstring tests. The following sections introduce these two kinds of testing and the entire testing framework, analyzing its advantages, disadvantages and efficiency.

2.1 The Unit Test Framework

The *unit test framework* Figure 1 on the next page supports the unit test type of software testing and the implementation of test cases for modules under test. The unit test framework also supports a way to execute – from one point, with one single command – all the test case implementations related to the modules to be tested. As the unit test framework is written in Python, it accepts only Python modules for testing.

By the type of the test case implementation (the test case method or the test case string pair), we distinguish between the unit tests and the docstring tests in the framework. The *unit test* is a mechanism handling the test case functions, whereas the *docstring test* processes test case string pairs. While performing unit tests and docstring tests the unit test framework calculates the .

The in the framework is the proportion of source code lines executed during the execution of all unit and docstring tests to all executable source code lines. The code coverage is not only calculated for the module explicitly under test, but for all modules indirectly executed during test. As can be seen in Figure 1 on the following page, the unit test framework is strongly related to the Python `unittest` and `doctest` standard libraries. There are two main modules `unittest.py` and `doctest.py`. The names of the modules disclose that the former one manages the unit tests and the latter one manages the docstring tests.

The integration of the unit test framework to the software development process is almost simple. For all modules being tested by the unit test framework, we only need to write the test case functions or test case string pairs according to the requirements of the Python standard unittest and doctest libraries. Both of the test case implementations may exist on all levels (module, class, method, function).

The second thing is to define which modules may be tested recursively. Since the module dependency tree is often extra large, the framework offers a possibility to control for which modules the code coverage should be calculated. We simply define the directories, which should be taken or on the contrary which should be ignored.

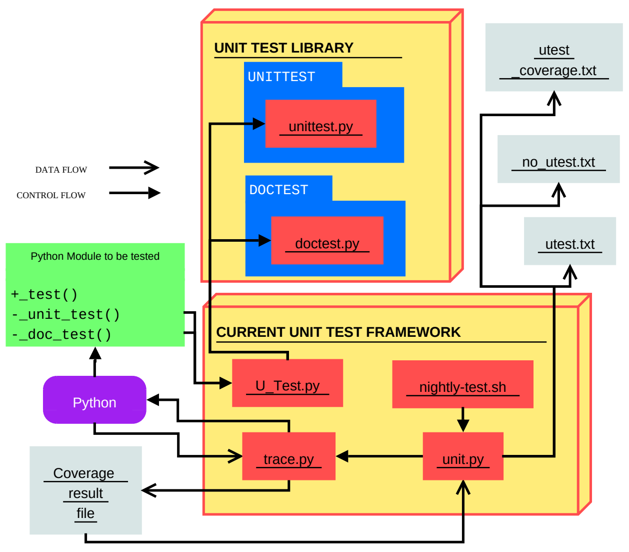

Figure 1: Current Unit Test Framework

The output of the framework are *coverage files* (`*.cover`) per tested modules each, then `utest_coverage.txt`, `utest.txt` and `no_utest.txt`. The latter three files are produced by `unit.py`. The `utest.txt` comprises the filenames of all modules being tested which have implemented some test cases, plus the result. The filenames of the rest modules being tested, which do not have implemented any test case are saved in the `no_utest.txt`. The additional information is the number of lines of code. The `utest_coverage.txt` stores all modules being tested. Further it provides for every filename the code coverage if it exists, the total number of lines and the total number of code lines. At the end of the `utest_coverage.txt` some statistics about the whole testing are added.

The coverage files are in fact the source files, but executed executable lines and not executed executable lines are distinguished with some marks. The marks for executed lines keep information how many times the source line was executed. These marks are needed especially for the calculations. The `trace.py` module provides further information printed on the output. But it is supressed in this framework.

In the previous lines we have described what the unit test framework is about and how we can integrate it into the development process. The necessary technical details like what must be called and what the control parameters are, are out of the scope of this document. Now we want to describe the concrete configuration of the unit test framework at *TT-Tech*. The modules to be tested (green in the Figure 1) are located in `/projects/SW/lib/python` directory. Only some of the modules have test case implementations. The framework creates coverage files in the same directory where the modules are located. The `utest.txt` and no_utest.txt are located at $ROOT_DIR. The utest_coverage.txt is located at $CASE_DIR. These two environment variables are defined in nightly-test.sh.

2.2 The Mechanism of the Unit Test Framework

The Figure 1 on the facing page shows us that the framework consists of three basic Python modules (unit.py,trace.py,U_Test.py) and one shell script. We will focus on the Python modules as they are the core of the framework. The shell script nightly-test.sh represents the single point of getting started the framework. The script is used mainly by the testers to start the nightly tests. It also checks out the required version of the modules to be tested and chooses a code coverage tool. In this case, it is trace.py.

Now we describe the main procedure of the framework. It begins in unit.py, which walks recursively through the given directory including the modules to be tested. We assume that some of the modules have test case implementations. Only these modules are directly tested by the framework. Further, we assume that the access point to these test case implementations is a function called _test(). This function uses unittest and doctest libraries directly or via U_Test.py module to execute all test case implementations. In fact, so it is with real Python modules at *TTTech*. When the unit.py finds such module, it applies the coverage tool to it. The coverage tool is implemented in trace.py. The trace.py lets another Python interpreter execute _test() in the given module. The new Python interpreter provides information needed for the calculations as well. U_Test.py is a small extension of the unittest.py and doctest.py, but it is not always used. The strategy

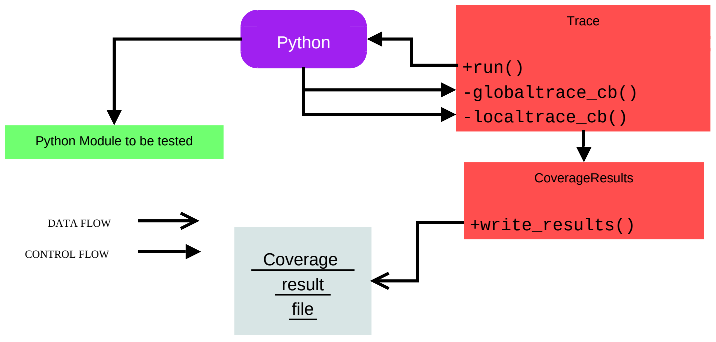

Figure 2: Code Coverage Calculation

of the coverage tool is based on debugging extension of the Python interpreter. Figure 2 depicts the relations between a module to be tested, the unit test framework and the resulting coverage file. The trace.py implements two important classes - Trace and CoverageResults.

The Trace class connects its callback functions to the tracing Python interpreter to get the trace information. The `globaltrace_cb()` callback function is called whenever the Python interpreter enters a function and the `localtrace_cb()` callback function is called before entering a new source code line. After the Python interpreter has finished the execution of the given module, the `Trace` class disposes of Python dictionary where positions of all traced code lines of all traced modules are saved.

This result is then used by the `CoverageResults` class which produces all coverage files. To produce the coverage file, first the original source code is compiled. From the compiled module we gain the information which lines in the source code are executable code lines. So, when the `CoverageResults` produces a coverage file, it knows exactly, which source code line is the executable one. The executable source code lines in the coverage file get marks in front of them to know which ones were executed and how many times and which ones were not at all. At the end the `unit.py` calculates the and some statistics stored in the files as already mentioned above.

2.3 The Analysis of the Unit Test Framework

If we look at the `unittest` library used by the framework more closely, we can realize that it enables us to implement the test cases in different ways (see `unittest` Python documentation). This variability is useful when the test case functions are already available, but not conformable with the library. On the other hand, if we write new test case functions, we may choose the class-way alternative.

Each test case function may be encapsulated in a class representing a test case implementation. This variability improves the flexibility of the framework. The question is whether the possible non-uniformity of writing test case implementations can harm the software testing or not.

The second thing worth analyzing are test case string pairs. The Python interpreter uses them as an input of a given interpreted module. Then the interpreter compares the output of the system with the given output stored in the test case string pair. The problem is that the received and expected results are compared at the character level. That makes the test case string pairs strongly dependent on the used Python interpreter version, thus worsening the flexibility of the framework.

The third important item is the calculation of the . As mentioned earlier, the is the ratio of the executed lines to all executable lines. The problem here is that the calculations depend on the developer's coding style, because if a statement stretches through more lines, the tracer evaluates it as more executable lines as well. This may lead to the fact that the results of two different developers are incomparable. That's why a norm, what a code line is, should be defined. If we inspect the code coverage algorithm, we can realize that the one and the same coverage file is generated many times. This problem comes into being, because the coverage file is generated not only for the direct tested module, but also for all indirectly used ones. The version of the coverage file on which the coverage result for the particular module is based is the one generated after the direct module test. So the coverage result is the coverage of the module's test case implementations. From the global point of view the coverage results do not concern the indirectly executed lines (i.e. the executed lines of the different modules). If all versions of one coverage file were merged and then used for the calculations, the results would be more real, since they would comprise all module's executions.

The fourth important point is the format of the output. The output of the unit test framework is produced in more text files placed in different places. We propose to research this aspect so that the really important data are available in a clear and sufficiently expressive form.

The last thing to be mentioned is the documentation. There is no manual that describes the usage of the framework precisely.

2.4 The Unit Test Mechanism

We have discussed the current unit test framework at *TTTech*. Now we shift us to Python `unittest` library, which is one of the two fundaments of the framework. First, we will talk about the unit tests Figure 3 and then in the further section about docstring tests Figure 5 on page 10, which is in `doctest` library.

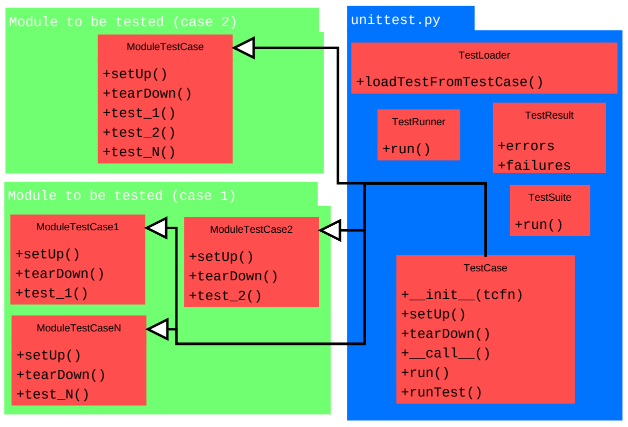

Figure 3: Unit Test Source Code Layout

The `unittest` library in `unittest.py` enables us to implement test case functions and then call them all from one location. There are two main ways how to connect test case functions to the `unittest` library. Before explaining these two cases we have to explain a `TestCase` class providing the connectivity. The `TestCase` class's main purpose is to encapsulate a test case function that overloades the `runTest()` method. The `run()` and `__call__()` methods of the `TestCase` then call this method. The next methods, the `setUp()` and the `tearDown()` methods are intended to establish and to close testing environment, called the *fixture*. What is important is the constructor of the `TestCase`. Its argument `tcfn` Figure 3 on the preceding page denotes a function that is called instead of the original or the overloaded `runTest()` method when calling `obj.run()` or `obj()`. This feature is used in the following text.

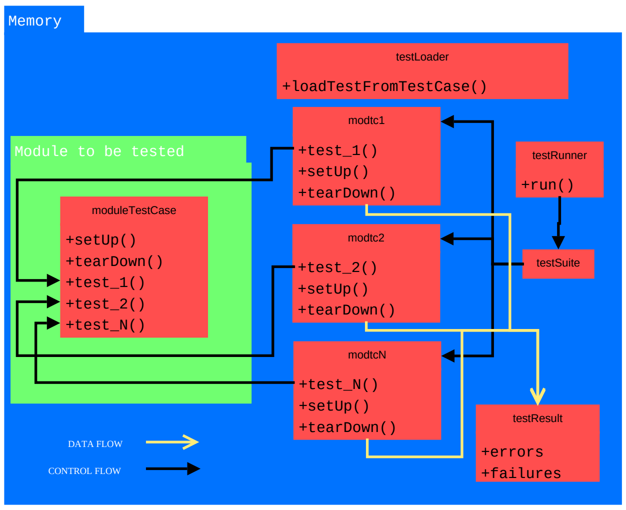

Figure 4: Unit Test Memory Layout (case 2)

In the first case (Case 1 in Figure 3 on the page before) we define a new `ModuleTestCase1..N` class for each test case function. The `ModuleTestCase1..N` classes are subclasses of the `TestCase` class then. The test case functions are implemented in (`test_1..N()`). The objects of the `ModuleTestCase1..N` classes are then executed. In each executed object the refering `test_1..N()` is executed.

In the second case (Case 2 in Figure 3 on the preceding page), we have a module's class `ModuleTestCase`, which comprises now all test case functions (`test_1..N()`). We cannot execute the `ModuleTestCase` object `moduleTestCase` Figure 4 as in the first case, because the `run()` method does not know which of the comprised `test_1..N()` test case functions should be executed. Therefore we create N copies of the `moduleTestCase` `modtc1..N` and for every object we denote one of `test_1..N()` test case functions to be called by the `run()` method. In this way we get virtually `TestCase` objects similar to the first case ones. For example, although the object `modtc2` comprises all `test_1..N()`, only `test_2()` will be executed upon calling `modtc2.run()` or `modtc2()`. The advantage of the second case is that the `setUp()` and the `tearDown()` are implemented only once, namely in the `ModuleTestCase`

class.
This transformation Figure 4 on the facing page is implemented in `TestLoader` method `loadTestFromTestCase()`. To understand it better a pseudocode Listing 1 is provided. The pseudocode is not exactly the same code as used in the `unittest` library.

Listing 1: Test Case Function Transformation

```
# in TestLoader class

def loadTestFromTestCase(ModuleTestCase)

 # finds all test_1, test_2, ..., test_N function names
 TCFuncNames=getTCFuncNames(moduleTestCase)

 for TCFuncName in TCFuncNames

  # create a copy of moduleTestCase
  # the test case function for this copy
  # has name TCFuncName
  newTestCase=ModuleTestCase(TCFuncName)

  # store the copy in a TestSuite object
  aTestSuite.add(newTestCase)

 return aTestSuite

# in TestRunner class,
# which calls all TestCase objects

def run(aTestSuite)
 for newTestCase in aTestSuite

  # newTestCase.run() calls setUp(), test_X()
  # and tearDown() functions
  newTestCase.run()
```

The further step - the same for both cases - is controlled by the `testRunner` object in the `run()` method. It runs the test suite the `TestSuite` which runs all `TestCase` instances where `test_1..N()` are called and it prints test statistics and error messages. When a test case is called and an error occurres, it stores the error messages to a given result instance of the `TestResult` type. This results are printed out after the `testRunner` finishes execution of the test cases.

2.5 The Analysis of the Unit Test

We could see that the `unittest` library is flexible as far as the implementation of the test cases concerns. On the other hand, the unit test as a software testing concept should almost perfectly test the software unit - the module, because it is specialized for that module. But there is not a control mechanism which makes sure, that the called test cases are the adequate tests. The quality of test cases lies on developers and not on testers. That's why the software may be tested successfully, but still it is not tested correctly and completely.

2.6 The Docstring Test Mechanism

The second part of the Python unit test framework Figure 1 on page 4 is the docstring test. The docstring architecture is depicted in Figure 5 on the next page. The docstring tests are similar to the unit tests. The difference between them is that the the test cases are implemented in the test case string pairs. One or more test case string pairs are implemented in documentation strings - the *docstring*s. The docstrings

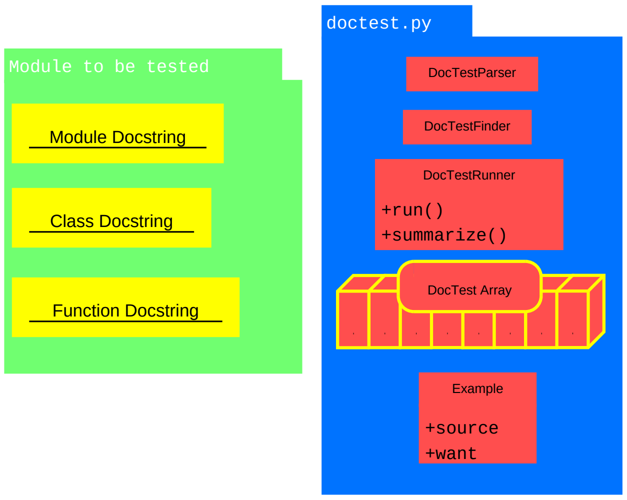

Figure 5: Docstring Test Source Code Layout

are special comments accessible dynamically. For every module, every class, every function and every method exactly one docstring may exist. The test case string pair coded in a docstring consists of a command line input and an expected output. The task of the `doctest` library implemented in `doctest.py` is to extract such docstrings from a given module, to execute the test case string pairs and to compare the received and the expected results.

At the begining the `doctest` library gets the module to be tested. The `docTestFinder` object Figure 6 on the facing page looks recursively for all docstrings in the module. The `docTestParser` object extracts all test case string pairs from a docstring. The `docTestFinder` then stores the pairs in the `example` objects each. The `example` objects of one docstring are collected in `docTest` object. The `docTest` objects are then stored in `DocTest` array `dta`. The `source` attribute in the `Example` class represents the command line input and the `want` attribute represents the expected result. When all docstrings are parsed the `docTestRunner` object executes all `example` objects and compares the results. The Figure 6 on the next page shows the structure of mentioned objects after all docstrings are parsed and test case string pairs are extracted.

2.7 The Analysis of the Docstring Test

The docstrings test is a simple and powerful software testing concept. This concept enables us to write the documentation and the tests of an item at once. That saves the time. A good feature of the docstring test (as well as that of the unit test) is the one which allows us to implement the test cases testing thrown exceptions. So we can implement both, the positive and negative test cases. The drawback of this concept

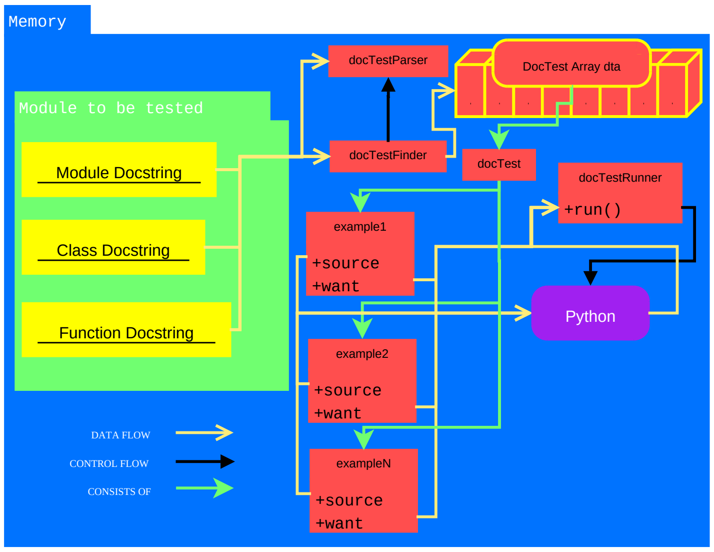

Figure 6: Docstring Test Memory Layout

is a strong dependence on the interpreter's version, especially when comparing traceback messages after an exception has been thrown. The dependence resides in a primitive result comparison, which is a string comparison. If the newer Python interpreter prints its traceback in a changed way, all touched test case string pairs in docstrings must be rewritten.

2.8 Summary

This chapter described the unit test framework currently used at *TT-Tech*. This framework uses two types of testing mechanisms, the unit test mechanism and the docstring test mechanism. The main difference is in the test case implementations for these two machanisms. The unit test mechanism uses the `TestCase` classes, whereas the docstring test uses for the implementation of the test cases the Python docstrings .

The current unit test framework provides two metrics: the line coverage and the code size measured in the SLOCs. For details, see Section 4.4.7 on page 37. The problem of the two metrics is that they are based on source code lines that are defined by the coding style of developers. As a consquence the metrics results of codes written by two different developers are not comparable.

The second weak point of the framework are the docstring tests where the given and expected results are compared with each other at the character level. Due to this low-level comparison, the framework strongly depends on the Python version used when comparing a Python traceback, whose format need not be backwards compatible.

The last weak point of the framework is the mechanism of how to collect metrics data. If a module is tested directly and indirectly via another module, then the metrics results are taken only from the direct testing method. In such a way the real metrics results differ from the calculated ones.

3 Overview of Testing Methods

This section covers testing methods, which can be classified according to the software developement stages at which those methods are used. Such classification is necessary, because this section focuses on testing methods used only at the implementation stage and because this thesis aims at improving code quality. Other development stages, such as analysis, design and delivery, have testing methods of their own. They are – in contrast to the testing methods used during implementation – rather informal, because they are based on vague user requirements. “Informal” means that these testing methods try to ensure informally that the specification and the design is correct and complete. At the design stage, however, the consistency of an abstract system model can be formally verified as well. On the other hand, the testing methods used at the implementation stage are rather formal and exactly specified, because they are based on the compiled code, the source code, or on both.

We try to classify and examine the testing methods used at the implementation stage to get an overview. There are several criteria according to which we can classify those methods. Almost all classification criteria are orthogonal to each other. Table 1 lists the classification criteria and the possible testing methods corresponding to those criteria. From now on, we use the term testing method instead of implementation stage testing method.

Criterion	**Testing Method**
Input	full, partition, random
Apriori Knowledge	white-box, black-box, gray-box
Test Control	automated, manual, code review
Test Nature	dynamic, static
Test Scope	function, module, component, system
Tested Items	functional, structural non-functional issues, difference-related issues

Table 1: Overview of the classification criteria

3.1 Criteria

The following sections explain and analyze the criteria listed in Table 1 on the preceding page.

3.1.1 The Input Criterion

The first classification is done according to the input data required by a testing method. There are three testing methods.

Full testing method: Requires all possible test cases to be used to test a program under test. This testing method is used rarely, because it is almost always impossible to test the program under test with all possible test cases. The problem is not only time, but also the finding of proper test cases.

Partition testing method: Tries to solve the problem of covering the whole input domain. This strategy is based on the similarity of test cases. Which test cases are similar depends on decision of software engineers. For example, if two different test cases produce the same control flow path, then they are similar. Similar test cases create a partition. This testing method then selects at least one test case from all partitions that are applied to the test.

Random testing method: As the name already says, the test cases are selected randomly.

Examine these three methods, we can exclude the full testing method, because it cannot be put into practice. Comparing the partition and random testing method one might say that the partition testing method is much more efficient than the random one. But the random testing method is under some circumstances more efficient than the partition testing method, because the random testing method is actually a partition testing method with one partition. In other words, the partition testing method randomly selects representative test cases from every partition, whereas the random testing method randomly selects representative test cases from the whole input domain (one partition). Thus, the difference between these two methods is the partitioning itself. When the input domain consists of a few large and many small partitions then the random testing method is statistically more efficient. The random testing method is also more efficient for detecting errors that occur rarely. [GS04][Gut99][Rus91]

3.1.2 The Apriori Knowledge Criterion

The second testing method classification is done according to additional knowledge about the program we have. When we have the interface specification (otherwise we could not use the program) we can also have the source code of the program.

White-box testing method: Requires knowledge about the internals of the system under test. This strategy checks not only the output but also the correctness of the internal steps.

Black-box testing method: Deals with interfaces only. The black-box testing method does not know the internal control and data flow. It knows only the input and output. The strategy is to check whether the output of a system is equal to the expected output that is mapped to the given input in the specification.

Gray-box testing method: Is a method that is in between two methods mentioned. It uses more than only interfaces (for example, in the black-box testing method), but less than all details (for example, in the white-box testing method. This testing method uses an abstract model that reveals some details of the system under test.

As the gray-box testing method is a trade-off between the white-box testing method and the black-box testing method, we need not examine it. The characteristics of the gray-box testing method are, in fact, a mixture of the characteristics of the other two testing methods.

The white-box testing method is used for the consistent testing of internal data structures and control flow. The advantage of the white-box testing method is that it can efficiently cover a huge number of test cases, because it is based on the principle of logical deduction. This means that we take logically independent test cases from which we can deduce that the remaining test cases will be successful if the former ones pass. To determine which test cases are logically independent we need to know the source code. To this end, we exclude logically redundant test cases.

A suitable example is a prime number generator. We cannot test the whole output, containing the interesting numbers, which are huge. The first barrier is that we need another prime number generator so that we can compare the results. The second barrier is the time needed to generate all prime numbers. Instead of examining the result, we test the generator for consistency by using the proven mathematical model. We only select test cases that test the control and data structures for correctness. For example, we test the prime number generator with small numbers. Then we can say that the generator is reliable. The white-box testing method is often called *structural testing method*, too. See also Section 3.1.6 on page 17.

The opposite of the white-box testing method is the black-box testing method mentioned above. The black-box testing method questions the principle of logical deduction. The problem is that if some specific test cases are successful then this does not imply that all other test cases pass successfully. The black-box testing method is also called *functional testing method* or *interface testing method*. See also Section 3.1.6 on page 17. [Rus91][Vig05][SL05]

3.1.3 The Test Control Criterion

The third criterion is controlling the testing process. There are three methods how to control a testing.

automated testing method: Means that the system under test is automatically fed with input data and that the output data are evaluated automatically. If the test cases defining the input data are generated automatically, then we call this **fully automated testing method**[AB81]. In the following, we do not differentiate between these two cases and use the automated testing methodology for both, unless explicitly distinguished.

Manual testing method: Means that the feeding of the input and the evaluation of the output is carried out by a human being.

Code review: Is no testing, but verification. A group of relevant developers review a source code and verify its correctness.

The advantage of the manual testing method is high flexibility and easy implementation. But the time to execute and examine all test cases can grow significantly. The advantage of the automated testing method is that the automated generation of test cases, the automated execution of testing and the automated examination of its results can reduce the time required for testing. The drawback of the method is that it is difficult to automatically generate test cases that cover the whole input domain, because a detailed formal specification is missing, although required. Sometimes the input domain is so huge that automated testing is impossible even with a detailed formal specification. Let us, by way of example, look at the testing of a graphical user interface. The second drawback is the relative long time to prepare a framework for automated testing. That is why a tradeoff between these methods is to be achieved.

As far as code review is concerned, this method can detect errors in the abstract levels of the program. On the other hand, typically fundamental errors can be overlooked. Therefore, more reviewers are recommended to cooperate with each other, when reviewing a source code.

3.1.4 The Test Nature Criterion

To differentiate between test and formal verification we have to introduce the test nature criterion.

Dynamic testing method: Is a testing method where the real system under test is executed.

Static testing method: Is not a testing method but a **formal verification**. Here, the real system under test is not executed. Testing is not verifying, and therefore this name static testing method is logically false. To be exact, we will use formal verification instead of the static testing method unless a method is a dynamic testing method. We can also use the same classification criteria names (for example, test scope rather than verification scope) for formal verifications, too.

The advantage of the dynamic testing method is that we can test the dynamic behavior of the system (race conditions, boundaries, communication, overload, etc.). The next advantage is that we can test real system, which will be applied somewhere. The disadvantage of this testing method is that we are almost never able to test everything. Sometimes we are not able to test all important issues. Such testing method is, for example, insufficient for fault-tolerant and security systems.

That is why we use formal verification, which helps to verify the correctnes of the abstract model (authentication protocol in a security system, or peak load behavior in fault-tolerant systems). Formal verification does not, of course, detect implementation errors.[Rus91][SL05]

3.1.5 The Test Scope Criterion

The next classification criterion is the test scope of the testing methods. The smaller the test scope, the easier and more reliable testing methods can be used, because the input domain of a smaller test scope is smaller than the input domain of a larger test scope. The monotony between the input domain size and the test scope size is violated when testing the interfaces, because of the encapsulation of details. For example, testing a system interface can be less complex than testing system component interfaces. The monotony is valid, for example, if the testing method is the structural testing method.

Summing up, we can say that testing methods for small scopes, such as statements and functions, used to be efficient and reliable. On the other hand, the interdependencies between such parts stayed unchecked.

3.1.6 The Tested Items Criterion

The last criterion classifies testing methods according to the items being tested.

Functional testing method: Focuses on items, such as interfaces functions, expressions and internal control flows.

Structural testing method: Focuses on data states and data flow.

Non-functional issues: Are issues, such as efficiency, performance, timeliness, etc.

Difference-related issues: Are all differences between two or more systems under test. It overlaps functional and structural testing methods, but the focus of this testing method is to make sure that none of the previous failures occurs again. [SL05]

We can compare the advantages and disadvantages of the first two testing methods at most. The third testing method cannot be compared, because it deals with the quality rather than the correctness of the program. The last testing method comprises functional and structural testing methods so that it need not be extra evaluated.

Comparing the efficiency of the first two methods is not so easy. A hint of which method to choose in a particular situation is given by the architecture of the program. There are two classes of software derived from the fact that software is code and data. Function-oriented software (for example, an internet phone program) and data-oriented software (for example, file system software). For the testing of function-oriented software we would choose one functional testing method, because it focuses on functions. For the testing of data-oriented software we would choose a structural testing method, because in that case the internal structure - filesystem must be correct. Often, both testing methods are used for testing a program. [Rus91][GS04]

3.2 A List of Known Testing Methods

Realistically spoken, we provide not only a general classification of testing methods, but also real testing methods. An overview of the classification of testing methods can be found in Table 3 on page 25. It is relevant here to define the difference between a testing method and a test. A *test* is a concrete implementation of a testing method, depending on the program to be tested, while a testing method is a strategy of what to test in which way. All tests described in the following are dynamic testing methods, with all verifications being implicit formal verifications (see Table 3 on page 25 and Table 4 on page 26 to get an overview).

3.2.1 Random Test

The name of this test determines its main characteristic, which says that the *random test* is a test for which its input domain subset is selected randomly. The distribution function of the selection may be adjusted according to the probability of failures occurring. [Rus91] The random test is mostly applied when an interface is to be tested. Therefore it belongs to the black-box testing method. Interfaces, mostly at lower abstract levels, are primarily tested in an automated way. So, the random test used to be an automated testing method as well. Random tests are rather used at lower abstract levels, because the lower level interfaces are rather simple and formally specified, which is a precondition for automated testing methods. As the random test used to test interfaces, it is also a functional testing method. As far as the scope is concerned, the random test may be used in every scope. [Rus91].

3.2.2 Regression Test

The fundamental idea behind the *regression test* is to use old test cases to prevent previous errors from occurring again. Such situations can appear when the program is changed in the maintenance phase or if a different program is developed, with a component used by this program needing to be tested as well. As the test cases used are available, the regression test is an automated testing method. It is also a black-box testing method, because only the test cases for black-box testing methodolgies remain in principle valid, if the program internals are changed or it is totally different. Sometimes regression testing can be also considered as a white-box testing method if the internals are not changed, but must be tested due to new unintended errors. With regard to test cases, the regression test is a partition testing method or random testing method. The regression tests may be speeded up by selecting relevant test cases instead of running all test cases. The test scope is in between functions and modules. The tested items are all except non-functional items. [Rus91][SL05]

3.2.3 Functional Test

A *functional test* is every test that tests any software component according to its functional specification. A functional test may be mapped to a couple of testing methods. It belongs to the black-box testing method, because of its testing interfaces. The functional tests can be mapped to the automated testing method and the manual testing method. It depends on the complexity of the specification of the interface under test. The decision of which method to choose often depends on the costs of each testing method. Furthermore, a functional test might be mapped to all testing methods mentioned in the input row and test scope row of Table 1 on page 13. [Rus91]

3.2.4 Structural Test

The *structural test* focuses on the internals of a program under test. There are two fundamental groups of structural tests. The first deals with control structures, such as decision points, loops and recursions. The second deals with data structures, such as variables, arrays, recursive lists, trees, etc. To implement structural test we need to have the source code of the program as an apriori knowledge.

The structural tests used to be applied in smaller rather than larger test scope, because the complexity of the internal structures of a system composed of many components used to be very high. To solve this problem we can transform the structural test from the white-box testing method to the gray-box testing method. Then we need an abstract model of the internals instead of the details of the source code. With regard to the input criterion and the control criterion, shown in Table 1 on page 13, a structural test can be every testing method. [Rus91]

3.2.5 Symbolic Execution

The *symbolic execution* is a test that constitutes a systematic technique for generating information about the inner workings of a program. [Rus91] This test is mainly suitable for complex expressions, such as recursive mathematical expressions. The test assigns initial symbolic values to all variables. Then it substitutes the variables in the expressions and continues with the simplification of the expressions. During and after the symbolic execution we can inspect the given expressions for correctness.

Symbolic execution is a white-box testing method, because it requires knowledge about the internal expressions. The test scope is a rather small test scope, such as a statement or function test scope. From the focus on symbolic execution it follows that the tested items are expressions that are functional items. The symbolic execution is suitable to be implemented as an automated testing method. As far as the input is concerned, all options are possible. [Rus91]

3.2.6 Program Spectra Analysis

To understand the program spectra analysis we first have to explain the *program spectrum*. The program spectrum, according to [HRWY98], characterizes or provides a signature of a program's behavior. There are the following six kinds of spectra (according to [HRWY98]). The *branch spectra* record the set of conditional branches that are exercised, as a program P executes. The *path spectra* record the set of loop-free intraprocedural paths that are traversed, as P executes. The *complete-path spectrum* records the complete path that is traversed, as P executes. The *data-dependence spectra* record the set of definition-use (see Section 4.3 on page 30) pairs that are exercised, as P executes. The *output spectrum* records the output produced by P, as it executes. The last spectrum, called *execution-trace spectrum*, records the sequence of program statements traversed, as P executes.

The main idea behind the program spectra analysis is to produce the program spectra for two or more programs, then to compare them and, finally, to locate where the programs behave differently to each other. To complete the description of this testing method we have to mention the substantial problems connected with it. The first one states that the difference between the program spectra of two compared programs P and P' does not imply that one of these programs fails where the program spectra differ. The second one states that if one program fails and another one not does not this implies that their program spectra must be different. For more details, see [HRWY98].

This method is a difference-related testing method, because we focus on differences between at least two similar programs (for example, an old program and a newly changed one). The extraction of program spectra information from the source code arranges this method to be a white-box testing method. As far as the test scope is concerned, it is suitable for a simple, smaller scope rather than for a complex, larger one. As the program spectra must be generated, it is supposed to be an automated testing method. As far as the input is concerned, it depends on the application.

3.2.7 Anomaly Detection

The *anomaly detection* is a formal verification method that evaluates the quality of a given source code. To detect anomalies the detection method must know the syntax and semantics of the source code. Typical anomalies are unused variables or unreachable statements.

This formal verification works with source code that has an exactly defined syntax, thus this verification can be automated. The scope of anomaly detection is almost unlimited due to its linear complexity. The verified items are structural items, such as control and data structures.

The main advantage of this formal verification method is the automated detection of locations where errors may reside, without understanding the program or its specification. [Rus91]

3.2.8 Walk-Through Verification

The *walk-through verification* is a code review method, because it is done by people. The relevant people inspect the source code systematically. They cover every piece of logic at least once and take every branch at least once. According to [Fag76] there are four important reviewer roles: moderator, designer, coder and tester. The coder reads the source code and the others walk through the code.

The advantage of this formal verification method is that there is no need for executable testing programs and formal specifications. This formal verification may be applied to earlier development stages, too. [Rus91]

The walk-through verification is like partition testing method because some specially selected input data are necessary for walking through. The verification scope is rather smaller, because it may be very impractical to walk through the whole system code. Verified items are both functional and structural items.

3.2.9 Mathematical Verification

The *mathematical verification* is another code review method that formally verifies abstract models for consistency. This can be done at various levels of formality. The process is based on formal, mathematical principles [Rus91]. The mathematical verification method has two weak points. The first is the possibility of a flaw occurring in the verification itself. The second is that the verification does not find problems originating from a bad specification, which does not meet the user requirements.

As far as our classification is concerned, this verification is based on the source code. The verification scope may vary, because it depends on the level of abstraction of the verified model. The verified items are expressions and functions. Mathematical verification does not need any input data, because it operates with symbols. [Rus91]

3.2.10 Executable Assertions

The *executable assertions* method is a formal verification and dynamic testing method. In the first case embedded executable assertions are proven that they are always satisfied by being passed by the locus of control without problems. In the latter case the executable assertions embedded in the program react by printing a message or throwing an exception, when they are broken.

This method enables developers and testers to watch the internal behavior of the tested program. It also helps to locate the error in the program more quickly.

As far as our classification is concerned, the executable assertions method is a typically manual testing/verification method, because the assertions are evaluated manually. In some circumstances it may be designed as an automated testing/verification method. The testing/verification scope is unbounded. The tested/verified items are functional and structural items. Also non-functional items, such as timeliness, can be tested efficiently if this method is a dynamic testing method. Necessary apriori knowledge is the source code so that we can implement the executable assertions, thus it is a white-box testing method.[Rus91]

3.2.11 Adaptive Test

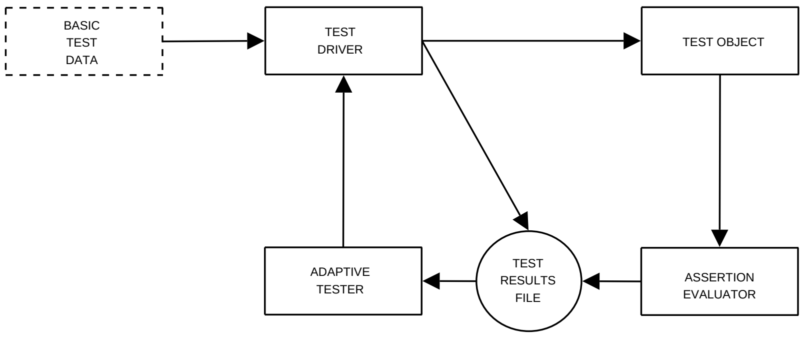

Figure 7: Adaptive Test Architecture [AB81]

The *adaptive test* is an automated testing method. The architecture of the adaptive testing method is shown in Figure 7. To understand the individual components, we have to explain the principle of the adaptive test. The first precondition is that the program under test must contain embedded executable assertions that trigger an error signal whenever they are evaluated to be false. The second precondition is `basic test data` that consists of some initial input data and their specifications, such as data range, minimum, maximum, etc. When both preconditions are fulfilled the adaptive test can start.

The `test driver` executes the test object with the basic test data. The `assertion evaluator` maintains a `test results file` where assertion violations are recorded together with their input data. Based on the `test results file`, the `adaptive tester` tries to find and generate such test cases that produce an as high as possible number of assertion violations. The `adaptive tester` uses an heuristic search algorithm for finding further necessary test cases. The classification of the adaptive testing method is shown in Table 2.

Criterion	**Testing Method**
Input	`basic test data`
Apriori Knowledge	black-box
Test Control	automated
Test Nature	dynamic
Test Scope	function, module, component, system
Tested Items	functional, structural

Table 2: Adaptive Testing Method Classification

The crucial point with respect to flexibility and robustness is the formal specification of the input data. With the adaptive test we can practically test every program for which input data can be represented as multidimensional vectors with bijective relation. Multidimensional vectors then create an input for an error function according to which the heuristic search algorithm seeks new test cases. The adaptive test is suitable for testing all tested item, with the exception of program differences Table 1 on page 13, because it works with data produced only during the current test. It can also be used for testing real-time and embedded systems, because the executable assertions may watch deadlines and resources usage. [AB81]

3.2.12 Object State Testing

The *object state testing* is a static testing method or, according to our convention, a formal verification method. It tries to find certain implementation errors whose detection would be much more difficult by using conventional structural and functional testing methods. Function-oriented software used to be modeled and tested using flat state machines, which tend to become excessively complex.
Therefore, the object state testing uses a *composite object state diagram* (COSD) that is a concurrent finite state machine composed of interacting *atomic object state diagram*s (AOSD). AOSDs are atomic finite state

machines. A substantial fact is that if we derive a spanning tree from the COSD, where nodes are the states and the edges are method calls, then we can find wrong paths (for example, disallowed sequences of method calls). Fo more details, see [KSGH94].

This formal verification method is like the white-box testing method, because it is based on the source code. It may also be like the gray-box testing method, because it requires at least an abstract model. The verification is done by using some input data. Dependening on the formalization of the correct and wrong paths in the COSD spanning tree, this verification can be carried out in an automated or manual way. The verification scope is the class or a set of classes, and the verified items are states of those classes. [KSGH94]

3.2.13 Unit Test

According to [Vig05] *unit test*s are all kinds of software unit tests that are done by developers. A *unit* can be a separate class that chains, through associations, classes or a module that contains a class.

The unit tests can be implemented as a white-box, gray-box or black-box testing method. Furthermore, it is an automated testing method. As far as the input data and tested items are concerned, they depend on the implementation of the unit tests only. The test scope of the test is defined by the unit definition.

3.2.14 Integration Test

The *integration test* is, in principle, similar to the unit test, with the exception of the test scope and the focus. The test scope of the integration test is components and system. The focus is on the communication among these system parts [SL05]. Sometimes it is hard to tell whether a test is an integration test or a lower-level test, because the borders are not sharp. [Lin05]

Testing Method	Criteria														
	Input			**Apriori Knowledge**			**Test Control**			**Test Nature**		**Test Scope**			
	full	partition	random	black-box	gray-box	white-box	automated	manual	code review	test	verification	function	module	component	system
Random Test			x	x			x			x		x	x	x	x
Regression Test		x	x	x	x	x	x			x		x	x		
Functional Test	x	x	x	x			x	x		x		x	x	x	x
Structural Test	x	x	x		x	x	x	x		x		x	x		
Symbolic Execution	x	x	x			x	x	x		x		x			
Program Spectra Analysis	x	x	x			x	x			x		x	x	x	
Anomaly Detection						x	x				x	x	x	x	x
Walk-Through Verification		x				x			x		x	x	x	x	
Mathematical Verification						x			x		x	x	x	x	x
Executable Assertions	x	x	x			x		x		x	x	x	x	x	x
Adaptive Test		x	x	x			x			x		x	x	x	x
Object State Testing		x			x	x	x	x			x	x	x	x	
Unit Test	x	x	x	x	x	x	x	x		x			x		
Integration Test	x	x	x	x	x	x	x	x		x				x	x

Table 3: Overview of the Classification of the Testing Methods and Formal Verifications

Testing Method	**Criteria**			
	Tested Items			
	functional	structural	non-functional	difference-related
Random Test	x			
Regression Test	x	x		x
Functional Test	x			
Structural Test		x		
Symbolic Execution	x	x		
Program Spectra Analysis				x
Anomaly Detection		x		
Walk-Through Verification	x	x		
Mathematical Verification	x	x		
Executable Assertions	x	x	x	
Adaptive Test	x	x		
Object State Testing		x		
Unit Test	x	x		x
Integration Test	x	x		x

Table 4: Overview of the Classification of the Testing Methods and Formal Verifications

3.3 Summary

This chapter examined a range of testing methods to get an overview of software testing and to find out which testing methods can be implemented in the new unit test framework. We found fourteen testing methods which we tried to classify. To do a reasonable classification of those methods, we focused on tests that are concrete implementations of testing methods. We found out that a test has six different aspects according to which we can classify that test. Thus we have also six criteria for the testing methods.

As far as the current unit test framework and this examination is concerned, we can see that the framework implements integration, regression and unit tests.

4 Research of Code Metrics

Section 3 on page 13 analyzes a range testing methods that are used in software testing for checking the correctness of the software developed. Choosing an efficient testing method depends on the type of application and on the effort of testing we want to expend. But that is not enough with respect to efficient testing. We also need to select the right test cases for the testing method chosen, otherwise the chosen testing method may become inefficient and, in the worst case, unreliable. To measure whether the selected test cases are sufficient or not we introduce , which require testers to select the right test cases.

4.1 Introduction

A *code coverage metrics* is a metrics that is based on a given code under test. Such metrics measures how many program structure items are covered by executing a test suite. A *test suite* is a collection of test cases. As the program structure is composed of many different items, we need a definition or criterion that determines what is an item for a particular test. That is why a metrics is in close relation to a criterion. In the following, we show some criteria on which metrics are based.

The *code complexity metrics* is similar to code coverage metrics, but it calculates the program code complexity using the program structure items instead of measuring the coverage. Again the program structure items must be defined. This kind of metrics is relevant as well, because it helps developers to keep the software smart and thus more reliable.

In this chapter, we list some code metrics, which are divided into three basic classes according to their criteria. The first class contains metrics whose criteria are based on the control flow. The second class contains metrics whose criteria are based on the data flow. Finally, the third class contains code complexity metrics suitable for measuring the complexity of the object-oriented program code.

4.2 Control Flow Code Coverage Metrics

The control flow are based on the control flow graph. The *control flow graph* is a graph as used in graph theory, with the vertices being linear sequences of computations and the edges representing control transfer between two nodes. Each edge is associated with a predicate that represents the condition of control transfer from node n_1 to node n_2. [ZHM97]

The precondition for using the control flow is the possibility to build a control flow graph from source code. For this reason, we need to detect and extract the following items:

1. statements

2. branches

3. conditions

The description of each metrics listed in the following contains a metrics definition and the practical limits of its usage.

4.2.1 Statement Coverage Metrics

The *statement coverage metrics* (SC) is the simplest metrics. The criterion is that each node in the control flow graph of the program code is executed at least once. The SC metrics can be easily implemented and used. On the other hand, SC is so weak that even some control transfers may be missed. [ZHM97][VB01]

4.2.2 Decision Coverage Metrics

The criterion for the *decision coverage metrics* (DC) requires that every edge of the control flow graph be contained in at least one control flow path covered by an executed test suite. This metrics is also called *branch coverage metrics*, because every outcome of each branch (see the Section 5.3.2 on page 45) is tested. This metrics is easy to implement and to use even for large programs. DC metrics does not require a complete test of each condition in each decision. [ZHM97][VB01][GS04]

4.2.3 Condition Coverage Metrics

The next metrics is the *condition coverage metrics* (CC). Its criterion requires that each statement be executed and that each condition in each decision take all possible values at least once. A *condition* is an atomic predicate or elementary Boolean expression that cannot be divided into further Boolean expresssions. At first glance, this criterion seems to satisfy the DC criterion. This is not true, because it is possible to omit such combinations of condition values that are necessary for a particular decision so that some decisions might not be covered. [VB01][GS04]

4.2.4 Decision Condition Coverage Metrics

The *decision condition coverage metrics* (DCC) defines a criterion that combines the DC criterion with the CC criterion. This means that a set of executions satisfies the DCC criterion if all the statements are covered, if all the decisions take all possible values and if each condition in each decision takes all possible values at least once. As this criterion satisifies the DC and CC, we say that it *subsumes* these two criteria. DCC also has another name – branch condition coverage metrics. [VB01][GS04]

4.2.5 Multiple Condition Coverage Metrics

The *multiple condition coverage metrics* (MCC) criterion requires that, for each decision, all combinations of condition values be covered by an executed test suite at least once. Thus it subsumes the DCC criterion. This criterion is unmanageable even for a moderate number of conditions, because the number of combinations of all the conditions grows exponentially. [GS04][VB01][ZHM97]

4.2.6 Path Coverage Metrics

The *path coverage metrics* (PC) defines a criterion that is satisfied if and only if an executed test suite covers all control flow paths in the program at least once. The difference between the PC criterion and the DC criterion is that the PC metrics requires all combinatiions of all decision outcomes. Keeping our metrics namings, this metrics should be called multiple decision coverage metrics. A test suite actually satisfies the PC criterion if it covers the entire control flow graph of the program. This code coverage criterion is the most complex criterion presented in this thesis. Satisfying this criterion may become infeasible for large software systems, because the number of decision combinations exponentially grows with the number of decisions. The PC criterion is also called all-pathscriterion. [ZHM97][Nta88][FW88][CPRZ85]

4.2.7 Modified Condition Decision Coverage Metrics

Although the MCC criterion subsumes a couple of metrics, which makes it very efficient for error detection, it is a practically unmanageable [GS04][VB01] criterion, as already discussed. The criterion for the *modified condition decision coverage metrics* (MCDC), which is in between the MCC criterion and the other criteria, has the same requirements as the criterion for DCC, with the addition that each condition in a decision is shown to affect the decision value independently of the other conditions. This is done by keeping fixed the values of all the other conditions in a decision while the variable condition affects the decision outcome. [GS04][VB01] [ZHM97]

4.2.8 Boundary-Interior Metrics

The *boundary-interior metrics* has two testing criteria, the boundary testing criterion and the interior testing criterion. The *boundary testing* criterion selects control paths that enter every loop body but do not iterate it, while the *interior testing* criterion selects control paths iterating every loop body several times. [Nta88]

4.2.9 Structured Path Coverage Metrics

The criterion for the *structured path coverage metrics* requires that all paths P of a given control flow graph be covered, where P does not contain any path p such that p consists of some subpath α, followed by more than k repetitions of some subpath s, followed by some subpath β. That means that each loop in the path is executed not more than k times. If the program under test is cycle-free then the structured path coverage metrics is equal to the path coverage metrics known as path testing metrics as well. [Nta88]

4.2.10 TER Metrics

The *test effectiveness ratios* (TER) is a class to which a testing strategy corresponds. TER_1 is segment testing (statement coverage metrics) and TER_2 is branch testing or decision coverage metrics. TER_{n+2} is satisfied if n LCSAJ's are tested. A *linear code sequence and jump* (LCSAJ) is defined in terms of the code text. An LCSAJ is a sequence of consecutive statements in a code text, starting at an entry point or after a jump and terminating with a jump or at an exit point. [Nta88][ZHM97]

4.3 Data Flow Code Coverage Metrics

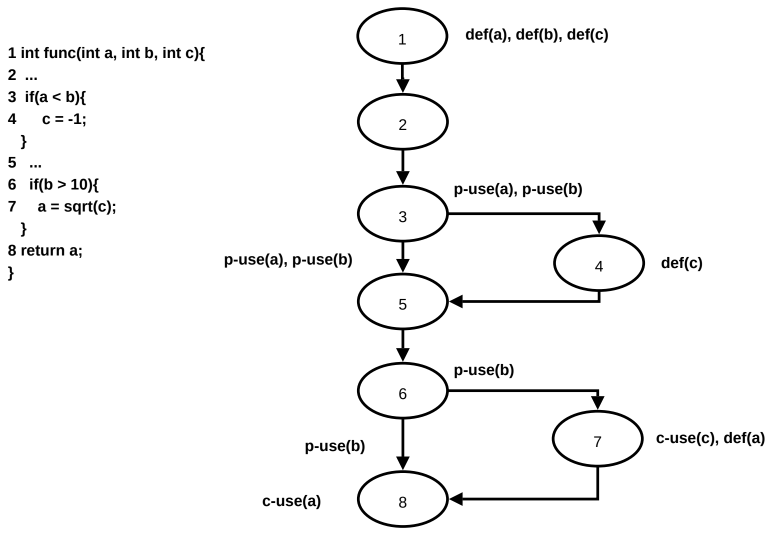
```
1 int func(int a, int b, int c){
2 ...
3  if(a < b){
4     c = -1;
  }
5  ...
6  if(b > 10){
7    a = sqrt(c);
  }
8 return a;
}
```

Figure 8: Data Flow Graph

The data flow are based on the data flow graph. The *data flow graph*, an example of which is shown in Figure 8, is a graph according to the graph theory, where the node represents a data processing, while the edge represents a control transfer between two distinct nodes [CPRZ85].

To understand data flow we need to introduce some definitions.

Generally, a *definition occurrence* `def(x)` is a storing of a value in a memory location of variable `x`, and a *use occurrence* `use(x)` is a fetching of a value from a memory location of variable `x`.

Use occurrence is divided into *computational use occurrence* `c-use(x)` and *predicate use occurrence* `p-use(x)`. The former affects the computation being performed or outputs the result of an earlier definition. The latter directly affects the control flow.

A `def(x)` *reaches* a `p-use(x)` or a `c-use(x)` if no redefinition of the variable `x` occurs along the path from the `def(x)` to the `p-use(x)` or the `c-use(x)`.

Another important definition is definition occurrence liveness. The definition occurrence of a variable *lives* at some location if there exists a control path from the definition occurrence of the variable to the location, where the variable is redefined yet. In other words, if a `def(x)` *reaches* a `use(x)`, then the `def(x)` is alive at the `use(x)`. [FW88][GS04][Nta88][Las82]

To use the data flow we need to be able to build a data flow graph from the source code. This means that we need to be able to detect and to extract the following items:

1. def statements
2. c-use statements
3. p-use statements
4. branches

Figure 9 on the next page shows the subsume relation between the data flow criteria. The subsume relation is explained in Section 4.2.4 on page 28. Path testing, branch testing (decision testing), and segment testing (statement testing) are not explained in this section, because they have been explained in the previous section. The following data flow metrics relate to one variable. If we want to apply any of these data flow metrics, we have to determine, which variables are to be dealt with.

4.3.1 All-P-Uses Metrics

The *all-p-uses metrics* defines a criterion that is satisfied by a test suite if for all `def(x)` all reachable `p-use(x)` are covered by the test suite at least once. [CPRZ85][Nta88][FW88][GS04]

4.3.2 All-Defs Metrics

The *all-defs metrics* defines a criterion that is satisfied by a test suite if for all `def(x)` at least one path from the `def(x)` to a reachable `use(x)` is covered by the test suite at least once. [CPRZ85][Nta88][FW88]

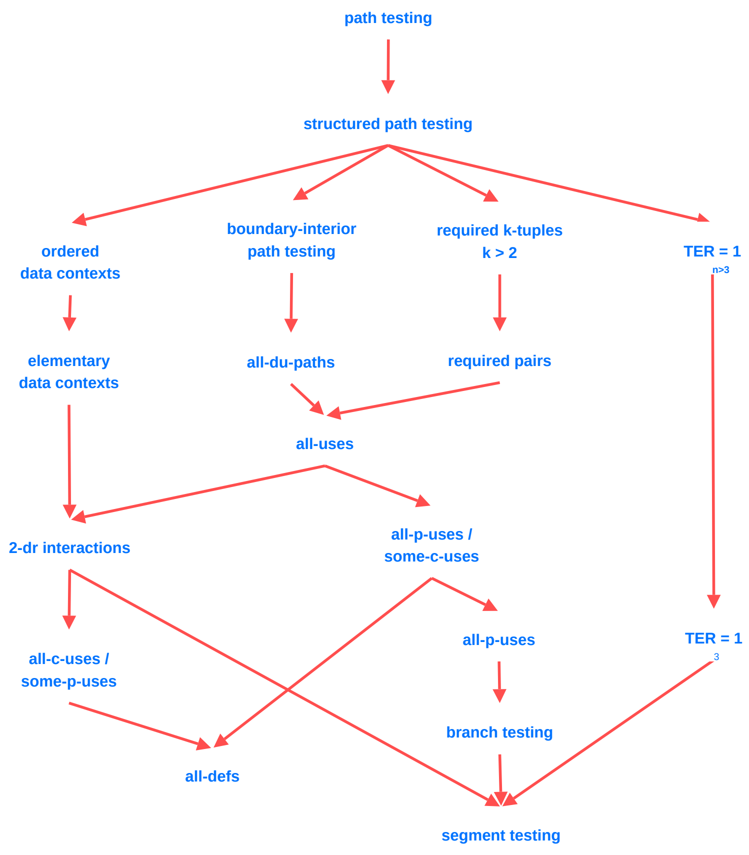

Figure 9: Partial Order of Data Flow Metrics [Nta88]

4.3.3 All-P-Uses Some-C-Uses Metrics

The *all-p-uses some-c-uses metrics* defines a criterion that is satisified by a test suite if for all `def(x)` at least one path from each `def(x)` to all its reachable `p-use(x)` is covered by the test suite. If no `p-use(x)` is reachable from the `def(x)`, then at least one path to a reachable `c-use(x)` must be tested at least once. [CPRZ85][Nta88][FW88]

4.3.4 All-C-Uses Some-P-Uses Metrics

The *all-c-uses some-p-uses metrics* defines a criterion that is satisified by a test suite if for all `def(x)` at least one path from each `def(x)` to all its reachable `c-use(x)` is covered by the test suite. If no `c-use(x)` is reachable from the `def(x)` then at least one path to a reachable `p-use(x)` must be tested at least once. [CPRZ85][Nta88][FW88]

4.3.5 All-Uses Metrics

The criterion of the *all-uses metrics* is satisfied by a test suite if for each `def(x)` at least one path to its every reachable `use(x)` is covered by the test suite at least once. [GS04][ZHM97] [Nta88][CPRZ85][FW88]

4.3.6 All-DU-Paths Metrics

The *all-du-path* criterion is satisified by a test suite if all possible paths between all `def(x)` and all their reachable `use(x)` are covered by the test suite at least once. [GS04][ZHM97] [CPRZ85][FW88]

4.3.7 2-DR Interaction Metrics

The *2-dr interaction* is a `def(x)/use(x)` pair, where the `use(x)` is reachable by the `def(x)`. The criterion is satisfied by a test suite if all 2-dr interactions in the code under test are covered by the test suite at least once. [ZHM97][Nta88][Nta84]

4.3.8 Elementary Data Context Metrics

An *elementary data context* (EDC) of an instruction is a set of all definitions living at the instruction. For node 8, shown in Figure 8 on page 30, the EDC is c. The criterion for the *elementary data context metrics* (EDCM) requires that all definitions in EDC must be covered by a test suite at least once. In other words, at least one subpath to the instruction must be covered. As shown in Figure 8 on page 30, one of $[1, 2, 3, 5, 6, 8]$, $[1, 2, 3, 5, 6, 7, 8]$, $[4, 5, 6, 8]$ or $[4, 5, 6, 7, 8]$ subpaths must be tested.

As the instruction is a very small and impractical unit in the currently used high-level programming languages, this metrics can be extended to a block of statements. The input of the block is a set of such variables that are, for the first time, used in the block, but defined before entering the block. The output of the block is a set of all variables, which are defined in the block at least once. EDC is sometimes called definition context as well. [CPRZ85] [Las82] [ZHM97] [Nta88]

4.3.9 Ordered Data Context Metrics

An *ordered data context* (ODC) of an instruction or a block is a sequence of all definitions living at that instruction. An *ordered data context metrics* (ODCM) requires that all ODCs of an instruction or a block be covered by a test suite at least once. Covering all sequences (ODCs) means covering all subpaths to the instruction or the block each of which comprises all living definitions. [CPRZ85] [Las82] [ZHM97] [Nta88]

4.3.10 Required Pairs Metrics

The *required pairs metrics* is similar to the 2-dr interaction metrics. It differs only in producing required pairs in branches and loops. If the `use(x)` of a 2-DR pair is `p-use(x)` in a branch, one required pair for each outcome of the branch is produced. For loops, two types of required pairs are considered: one exits the loop at the first opportunity, while the other asks that the loop be iterated several times. [Nta88]

4.3.11 Required k-tuples Metrics

The *k-tuple* or *k-dr interaction* is such a sequence as a sequence in 2-dr interaction, but expanded to k-nodes {}.

$$\begin{gathered} K = [\{def_1(x_1)\}, \\ \{use_2(x_1), def_2(x_2)\}, \\ \ldots \\ \{use_{k-1}(x_{k-2}), def_{k-1}(x_{k-1})\}, \\ \{use_k(x_{k-1})\}] \end{gathered}$$

The criterion is that all j-dr interactions $2 \leq j \leq k$ must be covered by a test suite. Note that a-dr interaction does not subsume b-dr interaction if $a \geq b$. [Nta88][ZHM97]

4.4 Code Complexity Metrics

Section 4.2 and 4.3 discussed code coverage metrics, which help to select the best test cases possible. This section focuses on *code complexity metrics*, which measure the complexity of code written mainly in object-oriented languages. The aim of the code complexity metrics is to decrease the number of potential errors in the code.

Before describing a couple of code complexity metrics we need to define the terms “coupling” and “cohesion” is. We use terminology used in object-oriented languages, especially terms like *class*, *subclass*, *inheritance tree*, *method* and *object* (or *instance*). *Coupling* between two classes means that at least one class acts upon the other, which, in other words, is any evidence of a method of one class using methods or instance variables of another class [CK94]. The *cohesion* expresses internal data dependence among the methods of a class. If a class has different methods that do not operate on any set of instance variables of the class or do not call other methods of the class, then the class is not cohesive [CK94].

To apply these metrics we need to be able to detect and extract the following items:

1. classes
2. subclasses
3. methods definitions
4. methods calls
5. class variables
6. class variables uses

4.4.1 Weighted Methods Per Class

The *weighted methods per class* (WMC) is defined as

$$WMC = \sum_{i=1}^{n} c_i$$

where n is the number of the methods in a class and c_i is the complexity of method$_i$. The complexity of the methods is not defined here in order to allow for the most general application of this metrics.

The WMC number shows us how many important methods a class contains. If the WMC number is high, then that class has probably many simple or a few complex methods. Such a class needs more development and maintenance effort and may have a greater potential impact on children, because they inherit all the methods of that class. Such a class also seems to be more application-specific and thus less reusable. [CK94]

4.4.2 Depth Of Inheritance Tree

The *depth of inheritance tree* (DIT) is, as defined in the graph theory, the depth of an inheritance tree built from classes, as defined in object-oriented languages. If multiple inheritance is supported, then the depth of the inheritance tree is the longest distance to the root of a class.

The higher the DIT number, the more complex is the class, since all the superclasses may influence its behavior. A high DIT number also tells us that the inherited methods have a larger reusability potential.[CK94]

4.4.3 Number Of Children

The *number of children* (NOC) is the number of immediate subclasses of a class. When the NOC number increases we have to be careful, because such class becomes more important and should therefore be better tested. A class with a high NOC number represents an important abstraction level, which hides specific details of the subclasses, thus the interface of that class has a larger impact on the system architecture. The NOC number also shows a high reusability of the class. In case of improper abstraction represented by such a class the whole system may become unusable or unmaintainable. [CK94]

4.4.4 Coupling Between Object Classes

The *coupling between object classes* (CBO) indicates the number of other classes to which a class is coupled (see above definition of coupling). The smaller the CBO number, the more independent and more reusable the class is. On the other hand, if the CBO number is high, then testing and maintenance of this class becomes more complex and more sensitive. [CK94]

4.4.5 Response For A Class

The *response for a class* (RFC) is the cardinality of the set of methods that can potentially be executed in response to a message received by an instance of the class. This set contains all methods of the class and all other methods called by these methods. As we can see, it is a level nesting of method calls. This limit is given due to practical considerations. The RFC number indicates that the class is more or less complex and thus needs more or less testing effort. [CK94]

4.4.6 Lack Of Cohesion In Methods

The *lack of cohesion in methods* (LCOM) is a count of method pairs whose similarity is zero minus a count of method pairs whose similarity is not zero (in case of a negative result, the result is adjusted to zero). The *similarity* of a method pair is defined as the number of instance variables on which both methods operate. The zero LCOM number means that the class may be cohesive, because there are a few (up to one half) or no method pairs operating on the same set of instance variables. On the other hand, if the LCOM number is positive, then the cohesion of a class is low.

The cohesiveness of methods within a class is desirable, because it promotes encapsulation. The lack of cohesiveness indicates that the class should be split into two or more subclasses, because it consists of two or more independent parts. Low class cohesion increases complexity and thus the likelihood of errors occurring during the development process. A low cohesive class warns that the class design is probably not correct. [CK94]

4.4.7 Code Size

One of the simpliest code complexity metrics is the *code size metrics*. Although it is such a simple code complexity metrics, it is not so easy to define what the smallest unit to be counted is. One way is to define a source code single line of code (SLOC) to be the smallest countable unit. The drawback of this model is that it is quite difficult to define what a single line of code is, because it depends on the developer's coding style. This is why we need to define a unit that does not depend on the developer's coding style. According to [Rus91] syntactic tokens should be the smallest countable units rather than the SLOCs. Then this metrics does not depend on the developer's coding style whithin a language. [Rus91]

4.5 Summary

In this chapter we presented metrics that try to measure the quality of test suites. There are three classes of metrics: the control-flow code coverage metrics, the data-flow code coverage metrics, and the code complexity metrics. Metrics in the first class measure how well a control flow is tested by a test. The second class of metrics measures the quality of how well a data flow is tested by a test. The last class of metrics concerns source code only and measures how efficiently the source code has been written.

The new unit test framework should support developers to write test suites having higher quality, thus some of the currently examined metrics must be chosen and integrated into the new framework.

5 The New Unit Test Framework

In this chapter we analyze how to eliminate the drawbacks of the current unit test framework mentioned in Section 2.3 on page 6. Also we propose some extensions and new metrics for the new unit test framework to improve the quality of testing the targeted software.

5.1 The Requirements and the Limitations Concerning Metrics

Before we discuss the design and the architecture of the new unit test framework we have to make clear which metrics can be implemented in Python language environment at all. Further we have to respect all requirements of the company (*TTTech*) concerning the new unit test framework.

5.1.1 The Company Requirements for the new Unit Test Framework

The company requires that the new unit test framework contains as many as possible metrics mentioned in the previous chapter. On the other hand the complexity of the metrics must be suitable for large nightly tests.

A further requirement is the possibility of measuring the execution time for each method. For this purpose, we introduce new metrics called *execution time per method* (ETM). This is an additional metrics that has not been discussed before, because it is not a . The ETM metrics provides two measured values per executed method. The first one is the gross execution time, which is the time between the entry point and the exit point of the method. The second one is the net execution time. The net execution time is calculated as the gross execution time minus the gross execution time of all methods and subroutines called by this method during an execution. Detailed explanation of the ETM is in Section 5.3.4 on page 47. The ETM should help to find the performance bottlenecks.

The last requirement is that metrics in the new unit test framework must be limited to the syntax level concerning the source code of the program. In other words, the metrics must not use information from the semantic level of the source code. These limitations keep the new unit test framework relative simple, thus suitable for larger systems under test. Of course, we need information from the dynamic analysis too. Exactly, we need to know, which parts of the program have been covered by the execution.

5.1.2 The Abstract Syntax Tree

To make choice, which metrics should be implemented in the new unit test framework, we need to analyze our infrastructure to gain necessary information. The first fundamental infrastructure is the Python *abstract syntax tree* (AST) delivered in the `compiler` and the `compiler.ast` modules included in the Python standard library.The AST data structure is the result of the parsing pass of the Python compiler. The AST provides almost all details defined in the Python grammar. See the documentation about the Python `compiler.ast` module where all possible AST nodes are described.

For us it is important to recognize and localize expressions, statements and compound statements from the source code, because each of the generated AST node represents a Python grammar structure linked with its location in the source code by a line number attribute `lineno`. See the Listing 2 and the Listing 3. The only exceptions are the abstract nodes like the statement block `STMT` node and the module node `MODULE` which do not store the line number information. We have to pay attention to the fact that AST data structure can vary from one version of Python interpreter to another. Due to this reason the new unit test framework uses the AST from the 2.5 version of Python only. The attributes with ("...") in the Listing 3 are in fact the references on some children nodes. They are blanked due to the better readability. For example, the `SUB` (Listing 3 at the line 15 has the `right` and the `left` attributes. They refer to the node `NAME` (line 16) and to the `CONST` (line 17).

Listing 2: Simple Parsed Python Source Code

```
a = 5
while a > 0 :
    a = a - 1
```

Listing 3: Abstract Syntax Tree in XML Form

```
1  <MODULE node="..." doc="..." lineno="None">
2   <STMT nodes="..." lineno="None">
3    <ASSIGN expr="..." nodes="..." lineno="1">
4     <ASSNAME flags="OP_ASSIGN" name="a" lineno="1"/>
5     <CONST lineno="1" value="5"/>
6    </ASSIGN>
7    <WHILE test="..." body="..." else_="..." lineno="2">
8     <COMPARE expr="..." lineno="2" ops="...">
9      <NAME name="a" lineno="2"/>
10     <CONST lineno="2" value="0"/>
11    </COMPARE>
12    <STMT nodes="..." lineno="None">
13     <ASSIGN expr="..." nodes="..." lineno="3">
14      <ASSNAME flags="OP_ASSIGN" name="a" lineno="3"/>
15      <SUB right="..." lineno="3" left="...">
16       <NAME name="a" lineno="3"/>
17       <CONST lineno="3" value="1"/>
18      </SUB>
19     </ASSIGN>
20    </STMT>
21   </WHILE>
22  </STMT >
23 </MODULE >
```

5.1.3 The Tracking System

The second infrastructure is the Python tracking system delivered also in Python standard library in the `sys` module. The tracking system provides a `line event` for every line of code about to execute, what is fundamental thing for tracking the Python source code. A very important fact is that if more statements are in the same line and only some of them are executed, the `line` event is still emitted. In other words, the tracking system is unable to recognize whether the whole line or only a part of the line has been executed when the `line` event has been emitted. See the Python source code in the Listing 4 on the next page in the line 7 and its output in the Listing 5 on the following page. The tracking system says that the line 7 was about to be executed, but we could not realize afterwards whether it was really completely executed or not. We can see that the line 7 was not executed, because the condition is false.

Besides this basic event there are three more events. Together, there are four events:

- `call event` emitted when a callable object is called,
- `return event` emitted when the return statement is executed,
- `exception event` emitted when the raise statement is executed and
- `line event` emitted when a line is to be executed.

The `call event` is a *global scope event*, because the scope is changed. The other events are *local scope event*s, because the scope remains. The scope changes whenever a function call happens. For example, when the `tst2 (2)` is executed at the line 8 then a `call event` is emitted. Next example is the return statement (line 19) where first the `line event` and then the `return event` are emitted. See the example in the Listing 4 on the next page and the tracking system results in the Listing 5 on the following page. Among other useful pieces of information these events provide the `line number`, the `module name` and the `function name`. For details see the Python documentation for the `sys` module.

The tracking system is sensitive to the Python version. During this thesis work we have found out that the track of the `while` compound statement differs in the Python 2.5 from the Python 2.4. The difference is the next executed line after the last statement in the `while` body, while the condition is true. In our example, the next executed line in the Python 2.4 would be the line 15 (the condition) instead of the line 16 (the first statement in the body) in the Python 2.5. We suppose that this line 15 is ignored in Python 2.5 because the line number stored in the compiled code possibly does not match against the real line number in the source code. This inconsistency is possibly caused by some optimization enhancements in Python 2.5.

Listing 4: Simple Tracker

```
 1  # For Python 2.5
 2  import sys
 3
 4  def tst1 () :
 5      a = 6
 6      b = 7
 7      if a == 8 : c = 2
 8      d = tst2 (2)
 9      if d == 0 :
10          raise StandardError ("exception␣raised")
11      d = 8
12
13  def tst2 (a) :
14      b = a
15      while a > 0 :
16          d = 7
17          a = a − 1
18          c = 6
19      return b % 2
20
21  def global_cb (frame, event, arg) :
22      print "global:␣event:␣%s,␣line␣%s" % (event, frame.f_lineno)
23      return local_cb
24
25  def local_cb (frame, event, arg) :
26    print "local:␣event:␣%s,␣line␣%s" % (event, frame.f_lineno)
27    return local_cb
28
29  try :
30
31      sys.settrace (global_cb)
32      tst1 ()
33
34  except StandardError, e :
35      sys.settrace (None)
36      print "%s:␣%s" % (e.__class__.__name__, str (e))
37  finally :
38      sys.settrace (None)
```

Listing 5: Simple Tracker Result

```
global: event: call, line 4
local: event: line, line 5
local: event: line, line 6
local: event: line, line 7
local: event: line, line 8
global: event: call, line 13
local: event: line, line 14
local: event: line, line 15
local: event: line, line 16
local: event: line, line 17
local: event: line, line 18
local: event: line, line 16
local: event: line, line 17
local: event: line, line 18
local: event: line, line 19
local: event: return, line 19
local: event: line, line 9
local: event: line, line 10
local: event: exception, line 10
local: event: return, line 10
StandardError: exception raised
```

5.2 The Arguments About the Metrics

In this section we explain, which metrics can be implemented in the new unit test framework generally. We also submit arguments, why we reject the other metrics.

5.2.1 The Chosen Control Flow Code Coverage Metrics

In the Section 4.2 on page 27 we discussed the control-flow code coverage metrics. We are going to choose all possible metrics keeping the given requirements and limitations. The first limitation lies in the fact that we have to extract the necessary structures from the result of the syntactic analysis only. The metrics, which need more information, like runtime values, are all metrics evaluating the conditions. These are the

CC, DCC, MCC and the MCDC metrics, thus they must be eliminated.

One of the company's requirements is keeping the complexity of metrics low. That is why we have to exclude all the path-based coverage metrics like the path coverage metrics, the structured path coverage metrics, the boundary-interior metrics and TER metrics.

The only metrics fulfilling all the requirements are the statement coverage metrics and the decision coverage metrics. Unfortunately, the tracking system is insufficient for both the SC and the DC metrics to produce the correct results. The problem is that the resolution of the tracking system is the line resolution only, what means that it hides the information how many statements in the same line have been really executed. Thus the coverage of some statements can be determined incorrectly. To prevent such situations we have to make a preliminary which checks the source code for multiple statements per line and warns the user.

5.2.2 The Rejection of The Data Flow Code Coverage Metrics

All data-flow code coverage metrics are based on the data flow graph (DFG) as mentioned in the Section 4.3 on page 30. To build the DFG we need the `def` and `use` statements. The Python language is a dynamic language what means that classes, function definitions and attributes can be defined in runtime. Therefore, extracting the real data flow graph within the syntax level is impossible, because we do not know where and when a `def` or `use` have occurred. Thus all the data-flow code coverage metrics have to be rejected.

5.2.3 The Problems With The Code Complexity Metrics

The difference between the former two groups of metrics and the code complexity metrics is that here we do not need any information gained from the tracking system. They are based on the generated AST only.

As we saw in the Section 4.4.1 on page 35 the WMC metrics can be implemented for Python source code if the complexity c is based on the AST, too. For example, c may be the number of `if`, `for` and `while` compound statements.

The next two metrics DIT and NOC are based on the class inheritance tree. To construct the class inheritance tree, we need the information about superclasses. This information is provided completely by the AST for every class definition, so DIT and NOC can be implemented using the information from AST.

The CBO metrics measures the coupling between classes. Again, we have to remember that Python is a dynamic language, thus we do not know (if based on source code only), how many classes really exist at runtime. That is why we cannot compute CBO from given AST.

The fifth metrics RFC has a characteristics similar to the CBO metrics, since it needs information about the number of methods both, defined and called. Because the methods may be added or removed dynamically, the RFC also cannot be computed from the pure AST data structure.

The sixth metrics the LCOM works with the similarity of methods. To calculate its value, we need the class attributes. And again, the class attributes can be added or removed dynamically and therefore to calculate the LCOM from the pure AST is impossible, thus the LCOM has to be rejected.

The last metrics is the best known the code complexity metrics - the code size metrics. The AST gives us a possibility to solve the problem with the SLOC unit. If we choose the AST node as a size unit, then the metrics is independent from the coding style. This metrics only requires the AST, so it can be implemented for the Python source code.

5.3 The Design of the Metrics

We have just investigated the metrics themselves and their limits. Now we are going to specify the design of the chosen metrics in the new unit test framework. Taking in account the limits originating in the Python language characteristics and the requirements of the *TTTech* company, we design a framework containing the following metrics:

1. Statement Coverage (SC)
2. Decision Coverage (DC)
3. Code Size
4. Execution Time per Method (ETM)
5. Weighted Methods per Class (WMC)
6. Response For a Class (RFC)

Further details about these metrics are dealt with in the following sections.

5.3.1 The Statement Coverage

The *statement* in the AST is defined as follows:

1. a direct child of any `<STMT>` node (Listing 3 on page 40)
2. a direct child of any `<IF>` or `<TRYEXCEPT>` node and at the same time the child is not `<STMT>`

The SC is calculated as a division of the number of covered statements by the number of all statements. Before calculating metrics we have to check whether maximum one statement is placed per line or not so that the result of the tracking system is reliable. The checker compares line numbers of the found `if` or `elseif` or `for` or `while` or `except` statements with the line numbers of their first body statements.

An exceptional case is the `else` statement ommitted in the AST. But that is not a problem, because a partial execution of such a line is impossible, due to the non-existence of the condition causing a partial execution of the line. The second exceptional case is the `except` statement where the name of the exception class and the name of its instance are defined. See the (Listing 6 line 3). According to the AST this line carries two statements, namely the `StandardError` and the `e`.

Listing 6: Two statements in a line

```
1 try:
2     hello ()
3 except StandardError , e :
4     print "%s" % e
```

Concerning the aggregation of SCs to one SC, the aggregation is defined as the number of all covered statements of all modules divided by all statements in all modules. Generally, the aggregation of the metrics is used for calculating the metrics of a system.

5.3.2 The Decision Coverage

To calculate DC we need to define and distinguish a decision from a branch point, because they need not be identical seeing the AST of a source code. A *decision* is a compound condition according to which the control flow is changed. A decision is represented by a set of AST nodes. The *branch point* is represented by a single AST node, at which the control flow is really branched. The branch point is more important for the DC metrics. The *branch* is a sequence of executed AST nodes, which begins with a branch point node.

To find branch points in a AST, we have to set two criteria for AST nodes that can be taken as branch points in the Python language:

- the node can have more than one successor nodes or
- the node can have at least one successor node, but it can be the last executed node.

We have searched for branch points in the AST generated from a source code that has contained all grammar constructs, which imply control flow branching. These grammar constructs are `if`, `while`, `for`, and `try-except` compound statements. All found branch points are the last nodes of statements marked as “branch point” in the Listing 7 on the following page, Listing 8 on the next page, Listing 9 on the following page and in the Listing 10 on the next page.

The DC is then defined as the number of covered branches divided by the number of all branches. Some branches are ignored, as they are implicitly counted in immediately following decision, because they are in a sequence. See the Listing 7. If the true branch of the decision 2 has been executed, then the false branch of the decision 1 must have been executed too.

Listing 7: IF Compound Statement

```
1  # Python 2.5
2  if a == 5 :   # branch point 1 (jump line 3 or line 4 — ignored branch)
3      b = 6
4  elif a == 6 : # branch point 2 (jump line 5 or line 7)
5      b = 7
6  else :
7      b = 0
```

This piece of code hides actually two decisions (the line 2 and 4), because it contains two compound conditions. Every decision has exactly two branches due to Boolean type of the condition result (`True`, `False`). Intuitively, this code has three alternatives (the line 3, 5 and 7) that may or may not be covered. So we would expect that the DC would evaluate these three alternatives. Unfortunately, we have two decisions, thus four alternatives. The DC results might be then $0\%, 25\%, 50\%, 75\%$ or 100% instead of $0\%, 33\%, 67\%$ or 100%. To achieve our intuitive expectations we count the true branches only and ignore the false branches of all decisions (the line 2 in the example), unless they are the last ones (the line 4 in the example).

The similar situation appears in the `try` - `except` construct in the Listing 8, because these decisions are in a sequence too.

Listing 8: TRY - EXCEPT Compound Statement

```
1  # Python 2.5
2  try :
3      a = 6
4      func ()          # exception may be raised here
5      b = 7            # branch point 1 (jump line 11 or line unreachable)
6  except Error1 , e : # branch point 2 (jump line 7 or line 8 — ignored branch)
7      print e
8  except Error2 , e : # branch point 3 (jump line 9 or out)
9      print e
10 else :
11     c = 8
```

To close this section, we depict the decisions of the rest compound statements `FOR` and `WHILE` in the Listing 9 and in the Listing 10.

Listing 9: WHILE Compound Statements

```
1  # Python 2.5
2  a = 8
3  while a > 0 :          # branch point 1 (jump line 4 or line 9)
4      d = 9
5      a = a — 1
6      c = 6              # branch point 2 (jump line 4 or line 9)
7                         #                (jump line 3 or line 9) in Python 2.4
8  else :
9      b = 7
10 x = a
```

Listing 10: FOR Compound Statements

```
1  #Python2.5
2  for a in range (10) : # branch point 1 (jump line 3 or line 5)
3      print a
4  else :
5      print "stop"
```

As for the aggregation of the DCs to one DC, the aggregation is defined as the number of all covered branches of all modules divided by all branches in all modules.

5.3.3 The Code Size

The code size metrics is very simple metrics, which counts the AST nodes. See Listing 3 on page 40 for Listing 2 on page 40. The aggregated code size is the sum of all code sizes.

The new unit test framework also supports the line coverage metrics. The *line coverage metrics* (LC) counts the covered lines instead of covered AST nodes. This metrics, which strongly depends on the coding guidelines is supported so that the user can compare the results of the new metrics with the results of the well-known line coverage metrics.

5.3.4 The Execution Time Per Method

We know two execution times - the gross time and the net time. The *gross time* is the time from any entry point of a method to any exit point of the method. The *net time* is the gross time without the gross times of all subroutines called by the method. See Figure 10.

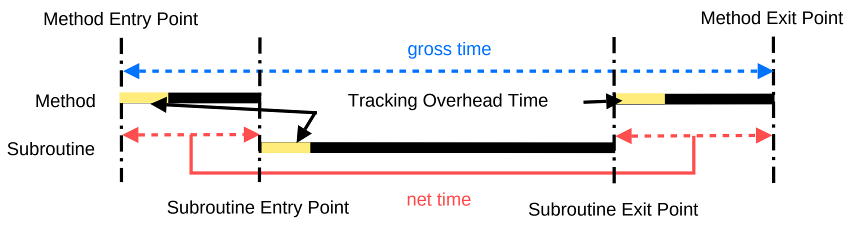

Figure 10: Gross Time and Net Time

To gain the current time instance we use the `time.clock()` function. Notice the yellow bars, which mean the tracking overhead time, the time of the callback functions triggered on the `call` and the `return` events. This time may be neglected if the subroutine takes enough time. But for short calls the overhead time influences the measurements negatively. Of course, each `line` event causes overhead time too.

The ETM on the system level is the sum of theETMs in the module level and the ETM in the module level is the sum of the ETMs in the function levels. Since the gross time must not be summed, the aggregated ETMs are net times only.

5.3.5 The Weighted Method Per Class

The WMC is in fact a special code size metrics, since it counts some predetermined AST nodes in the class methods only. That means for example that the top level functions and the expressions are ignored. It counts the occurrences of all binary, compare and arithmetic operators, the occurrences of all compound statements and the occurrences of all nested classes and functions. Although the nested classes and the functions are parsed recursively, their results are separated from the results of the enclosing methods and classes. The system WMC is the sum of the module WMCs.

5.3.6 The Response For A Class

In Section 5.2.3 on page 43 we have analyzed that RFC is impossible to be implemented for Python language. But if we do some restrictions we can approximate to the real RFC. We need one restriction only, namely we count methods that are explicitly defined in explictly defined classes. The less dynamic definitions of methods and classes in the source code the better the results of the RFC.

The RFC metrics is similar to the WMC metrics, since it counts some predetermined AST nodes too. The predetermined AST nodes are the classes, the functions and the function calls. Thus the RFC handles with the classes and the functions (nested as well) in the same way as the WMC. The system RFC is the sum of the module RFCs.

5.4 The Input and The Output of The New Unit Test Framework

5.4.1 Framework Command Line Input

This section shows options provided by the new unit test framework. The main module of the framework is the `Unit_Test_FW_Tool.py`, which accepts the options. After the options a list of start modules and directories is expected. Every start module must contain some unit tests or docstring tests, otherwise it is ignored. In case of the start directory all modules are executed recursively as the start modules.

Unit_Test_FW_Tool.py [OPTIONS] (start_module|start_directory)+

This is the synopsis of the new unit test framework main module `Unit_Test_FW_Tool.py`. The options are described below. The `start_module` is any Python source code module. If the module does not contain any unit tests or docstring tests the module is not executed completely, since the framework starts with such tests only. If a `start_directory` is given, then the directory is traversed recursively and loads each found Python source code module as a `start_module`.

--ignore, -i path(:path)*

Defines directories where all modules are completely ignored. This option is recursive, thus all modules in all subdirectories are ignored too. This option has higher precedence than the start modules and directories. The main reason for this option is to exclude the standard Python libraries to prevent them from evaluation.

--output, -o global_xml_file

The global results are placed in the standard output or in the file `global_xml_file` if this option is given. The suffix built from the `_label` string and the `.cover.xml` file name extension is added to the `global_xml_file`. For example the name of a global XML file can be `global_3.0.cover.xml`. The `_label` infix is described in the `--label` option.

--details, -d

This option forces the framework to provide details of metrics for manual checking of metrics results. All details are written in module XML file.

--debug, -g

The debug option writes module's AST and all tracked path in the module in the module XML file.

--label, -l

The label is used to distinguish the test results from other test runs. This label is contained in every module XML file and global XML file. It also codetermines the names of global and module XML files. Thus the result XML files from previous test runs are not overwritten. For example if the label is `3.0-test` then all modules `modname.py` have name `modname_3.0-test.cover.xml`.

--version, -v

Prints the version of this framework.

--verbose, -b

In the verbose mode each phase is announced. There are three phases: tracking, parsing and aggregating. In the tracking phase every start module is written to standard output. The letter "T" at the beginning informs us that we are in the tracking phase. The tracking phase has 4 result states. The result states are written after the start module name being tracked. The possible results states are: "OK" - the test starting with the start module has passed errorless. "NO TESTS" - the same as the first one, but no unit tests or docstring tests have been found. "FAILED" - an error has occurred. In the parsing phase, all modules being parsed and evaluated are written to the standard output. The letter "P" at the beginning informs us that we are in the parsing phase.

5.4.2 Framework XML Output

The output is divided into:

comand line, where errors are written if XML output is unavailable. Also the progression is written here if verbose mode enabled.

module XML file, which contains all results related to module. The name of the file is `module.cover.xml` if the Python module calls `module.py`. A content of module XML file is depicted in Listing 11.

Listing 11: Module XML File

```
<TEST date="06-Feb-07" module="Example.py" version="test␣V1.0" time="15:51:27">
 <METRICS>
  <SC covered="4" total="9" result="44">
   <SCDETAILS covered="False" line="20" id="2"/>
   .
   .
   .
   <SCDETAILS covered="True" line="30" id="26"/>
  </SC>

  <DC covered="9" total="12" result="75">
   <DCDETAILS taken="True" line="10" nottaken="-"     id="22"/>
   <DCDETAILS taken="True" line="12" nottaken="True"  id="28"/>
   <DCDETAILS taken="-"    line="31" nottaken="False" id="80"/>
   <DCDETAILS taken="True" line="31" nottaken="False" id="81"/>
  </DC>

  <SIZE result="34"/>

  <ETM netmin="0.001s" netmax="0.001s" module="Example.py" netavg="0.001s">
   <ETMFUNCTION function="test_a2"
                netmin="0.000s" netavg="0.000s" netmax="0.000s"
                grossmin="0.002s"  grossavg="0.002s" grossmax="0.002s">
    <ETMDETAILS gross="0.00189995765686" net="0.000440835952759"/>
   </ETMFUNCTION>
   <ETMFUNCTION function="test_a1"
                netmin="0.000s" netavg="0.000s" netmax="0.000s"
                grossmin="0.003s" grossavg="0.003s" grossmax="0.003s">
    <ETMDETAILS gross="0.00296688079834" net="0.00041389465332"/>
   </ETMFUNCTION>
  </ETM>

  <WMC wmc="2">
   <WMCCLASS wmc="2" class="utClass">
    <WMCDETAILS function="test_a2" c="1"/>
    <WMCDETAILS function="test_a1" c="1"/>
   </WMCCLASS>
  </WMC>

  <RFC rfc="6">
   <RFCCLASS rfc="6" class="utClass">
    <RFCDETAILS function="test_a2" calls="3"/>
    <RFCDETAILS function="test_a1" calls="3"/>
   </RFCCLASS>
  </RFC>

  <LC covered="2" total="3" result="67">
   <LCDETAILS covered="True" line="1"/>
   <LCDETAILS covered="True" line="3"/>
   <LCDETAILS covered="False" line="4"/>
  </LC>

 </METRICS>

 <TRACKER>
  <FILE name="Example.py">
   <PATH number="0">
    <LINE number="26"/>
    <LINE number="27"/>
   </PATH>
   <PATH number="1">
    <LINE number="29"/>
    <LINE number="30"/>
   </PATH>
  </FILE>
 </TRACKER>

 <PARSER>
  <MODULE node="..." count="0" end="False" name="Example.py" doc="..."
          next="1" lineno="None" successors="[]" id="0" predecessors="..."
          previous="None">
   <STMT count="0" end="False" parent="..." next="2" lineno="None" nodes="..."
          successors="[]" id="1" predecessors="..." previous="0">
    <CLASS count="0" code="..." end="False" name="utClass" parent="..."
          doc="..." next="5" bases="[Getattr(Name('unittest'),␣'TestCase')]"
          lineno="24" successors="[]" id="4" predecessors="..." previous="3">
     .
     .
     .
    </CLASS>
   </STMT>
  </MODULE>
 </PARSER>
</TEST>
```

Listing 12: The DTD of The Module XML File

```
<!DOCTYPE MODULEXMLFILE [

<!ELEMENT DC (DCDETAILS∗)>
<!ELEMENT DCDETAILS EMPTY>
<!ELEMENT ETM (ETMFUNCTION+)>
<!ELEMENT ETMDETAILS EMPTY>
<!ELEMENT ETMFUNCTION (ETMDETAILS∗)>
<!ELEMENT FILE (PATH+)>
<!ELEMENT LC (LCDETAILS∗)>
<!ELEMENT LCDETAILS EMPTY>
<!ELEMENT LINE EMPTY>
<!ELEMENT METRICS (SC|DC|ETM|SIZE|WMC|RFC|LC)∗>
<!ELEMENT PARSER ANY>
<!ELEMENT PATH (LINE+)>
<!ELEMENT RFC (RFCCLASS∗)>
<!ELEMENT RFCCLASS (RFCDETAILS∗)>
<!ELEMENT RFCDETAILS EMPTY>
<!ELEMENT SC (SCDETAILS∗)>
<!ELEMENT SCDETAILS EMPTY>
<!ELEMENT SIZE EMPTY>
<!ELEMENT TEST (METRICS,(TRACKER,PARSER)?)>
<!ELEMENT TRACKER (FILE+)>
<!ELEMENT WMC (WMCCLASS∗)>
<!ELEMENT WMCCLASS (WMCDETAILS∗)>
<!ELEMENT WMCDETAILS EMPTY>

<!ATTLIST DC covered CDATA #REQUIRED
             total CDATA #REQUIRED
             result CDATA #REQUIRED>
<!ATTLIST DCDETAILS line CDATA #REQUIRED
                    id CDATA #REQUIRED
                    taken (True|False|—) #REQUIRED
                    nottaken (True|False|—) #REQUIRED>
<!ATTLIST ETM module CDATA #REQUIRED
              netmin CDATA #REQUIRED
              netavg CDATA #REQUIRED
              netmax CDATA #REQUIRED>
<!ATTLIST ETMDETAILS gross CDATA #REQUIRED
                     net CDATA #REQUIRED>
<!ATTLIST FILE name CDATA #REQUIRED>
<!ATTLIST LC covered CDATA #REQUIRED
             total CDATA #REQUIRED
             result CDATA #REQUIRED>
<!ATTLIST LCDETAILS covered (True|False) #REQUIRED
                    line CDATA #REQUIRED>
<!ATTLIST LINE number CDATA #REQUIRED>
<!ATTLIST PATH number CDATA #REQUIRED>
<!ATTLIST RFC rfc CDATA #REQUIRED>
<!ATTLIST RFCCLASS rfc CDATA #REQUIRED
                   class CDATA #REQUIRED>
<!ATTLIST RFCDETAILS function CDATA #REQUIRED
                     calls CDATA #REQUIRED>
<!ATTLIST SC covered CDATA #REQUIRED
             total CDATA #REQUIRED
             result CDATA #REQUIRED>
<!ATTLIST SCDETAILS covered (True|False) #REQUIRED
                    line CDATA #REQUIRED
                    id CDATA #REQUIRED>
<!ATTLIST TEST date CDATA #REQUIRED
               module CDATA #REQUIRED
               version CDATA #IMPLIED
               time CDATA #REQUIRED>
<!ATTLIST WMC wmc CDATA #REQUIRED>
<!ATTLIST WMCCLASS wmc CDATA #REQUIRED
                   class CDATA #REQUIRED>
<!ATTLIST WMCDETAILS function CDATA #REQUIRED
                     c CDATA #REQUIRED>
]>
```

- **<DC>** - the decision coverage metrics

  covered - the number of covered branches

  total - the number of all branches

  result - the fraction of (covered/total) in percent

- **<DCDETAILS>** - the details about the decisions (if details enabled)

  line - the line position of a decsion

  id - the identification number of the decision in the extended AST.

  taken - whether the true branch was taken. If taken = "-" then this branch is ignored.

  nottaken - whether the false branch was taken. If nottaken = "-" then this branch is ignored.

- **<ETM>** - the sum of the execution times per method in the module

  module - the name of the module

  netmin - the minimum net execution time of the module

  netavg - the average net execution time of the module

  netmax - the maximum net execution time of the module

- **<ETMDETAILS>** - the details about the net times and the gross times per measurement (if details enabled)

  gross - the gross time

  net - the net time

- **<ETMFUNCTION>** - the execution times per method metrics

  function - the name of the method

  netmin - the minimum net execution time of the module

  netavg - the average net execution time of the module

  netmax - the maximum net execution time of the module

  grossmin - the minimum gross execution time of the module

  grossavg - the average gross execution time of the module

  grossmax - the maximum gross execution time of the module

- **<FILE>** - the block of all tracked paths <PATH> in a module (if debug enabled)

  name - the name of the module

- **<LC>** - the line coverage metrics

  covered - the number of covered code lines

  total - the number of all code lines

  result - the fraction of (covered/total) in percent

- **<LCDETAILS>** the details about the covered lines (if details enabled)

  covered - the coverage of a line (True/False)

  line - the line number

- **<LINE>** - the tracked line (if debug enabled)

  number - the line number

- **<METRICS>** - the block of metrics

- **<PARSER>** - this block of the parser subsystem debug information (if debug enabled)

- **<PATH>** - the block of lines <LINE> composing a tracked path (if debug enabled)

  number - the path number

- **<RFC>** - the sum of all response for a class metrics in the module

rfc - the sum

- **<RFCCLASS>** - the response for a class metrics

  rfc - the metrics result value
  class - the name of the class
- **<RFCDETAILS>** the details about the rfc value (if details enabled)

  function - the function name
  calls - the number of calls of the function
- **<SC>** - the statement metrics

  covered - the number of covered statements
  total - the number of all statements
  result - the fraction of (covered/total) in percent
- **<SCDETAILS>** - the details about the statements (if details enabled)

  covered - the coverage of a statement (True/False)
  line - the line number of the statement
  id - the identification number of the root node (extended AST node) of the subtree, which represents the statement
- **<SIZE>** - the code size metrics

  result - the number of extended AST nodes in the module
- **<TEST>** - the top block

  date - the date of performing the test
  module - the name of the module under test
  version - the label of the test
  time - the time of performing the test
- **<TRACKER>** - the block of the tracker subsystem debug information (if debug enabled)
- **<WMC>** - the sum of the weighted method per class metrics

  wmc - the sum
- **<WMCCLASS>** - the weighted method per class metrics

  wmc - the metrics result value
  class - the name of the class
- **<WMCDETAILS>** - the details about the wmc value (if details enabled)

  function - the name of the method
  c - the wmc complexity of the method

The module XML file is enclosed by the `<TEST>` tag and divided into three parts.

The first one is the metrics part enclosed by the `<METRICS>`, which contains all metrics results. See the Listing 11 on page 50 and the detailed descriptions of the tags above. To support the user to make the manual check for the correctness of the results the new unit test framework provides the `<...DETAILS>` tags for every metrics.

The second and the third parts are enabled if the debug option is enabled. The `<TRACKER>` contains the independent paths (see the Section 5.5.1 on page 60) for this file. The `<PARSER>` contains the extended AST for this file. See the Section 5.5.2 on page 61 for details.

global XML file, which contains aggregated results of all tracked modules in `<METRICS>`. It also contains the standard output of the system under test located in the `<OUTPUT>` tag. Further it contains the output of the docstring tests in the `<DOCSTRINGTEST>` tag and the unit tests output in the `<UNITTEST>` tag. The error messages are in the `<ERRORS>` tag. Each error message `<ERROR>` has the error type (`type`) attribute. The values of the attribute can be:

- *external* - the errors originating from the system under test
- *internal* - the errors from the new framework itself (should not happen).
- *parser* - the errors while parsing a module
- *tracker* - the errors originating from the tracking system itself (e.g. event callbacks)
- *xml* - the errors from XML subsystem
- *io* - standard input/output errors

Further it has module name attribute (`module` or `start`), where the error message comes from. The former attribute is name of the module where the error has occurred, whereas the latter one is the start module, from which the test has started (start module is the module, which the unit tests and docstring tests are started from). The reason for the `start` attribute is that framework does not know always, where the error has occurred. So it provides at least this attribute. For details of the global XML file structure see the Listing 13 on the facing page and the description of all tags.

Listing 13: Global XML file

```

<TEST date="10−Mar−07" version="v1.0" time="12:50:48">
 <MODULES tracked="2" parsed="2">
  <MODULE name="Module1.py"/>
  <MODULE name="Example.py"/>
 </MODULES>
 <METRICS>
  <SC covered="40" total="41" result="97"/>
  <DC covered="9" total="12" result="75"/>
  <SIZE result="137"/>
  <ETM netmin="0.006s" netmax="0.006s" netavg="0.006s"/>
  <WMC wmc="16"/>
  <RFC rfc="13"/>
  <LC covered="40" total="41" result="97"/>
 </METRICS>
 <DOCSTRINGTEST>
  DOCTEST: failures: 0
           tries: 4

 </DOCSTRINGTEST>
 <UNITTEST>
  ..
 −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
  Ran 2 tests in 0.005s

  OK

 </UNITTEST>
 <OUTPUT>
  333
  4
  000
  Error 3

 </OUTPUT>
 <ERRORS>
  <ERROR type="warning" module="Module1.py">
   Same−line warning (IF     statement) at line 10 in Module1.py
  </ERROR>
 </ERRORS>
</TEST>
```

The last part is the `<MODULES>` part and shows all parsed (tracked too) modules. Its attributes state how many modules have been tracked and parsed. In normal case, both of them should be equal.

Listing 14: The DTD of The Global XML File

```
<!DOCTYPE GLOBALXMLFILE [

<!ELEMENT DC EMPTY>
<!ELEMENT DOCSTRINGTEST (#PCDATA)>
<!ELEMENT ERROR (#PCDATA)>
<!ELEMENT ERRORS (ERROR)*>
<!ELEMENT ETM EMPTY>
<!ELEMENT LC EMPTY>
<!ELEMENT METRICS (SC|DC|ETM|SIZE|WMC|RFC|LC)>
<!ELEMENT MODULE EMPTY>
<!ELEMENT MODULES (MODULE)+>
<!ELEMENT OUTPUT (#PCDATA)>
<!ELEMENT RFC EMPTY>
<!ELEMENT SC EMPTY>
<!ELEMENT SIZE EMPTY>
<!ELEMENT TEST (MODULES,METRICS,DOCSTRINGTEST,UNITTEST,OUTPUT,ERRORS)>
<!ELEMENT UNITTEST (#PCDATA)>
<!ELEMENT WMC (WMCCLASS*)>

<!ATTLIST DC covered CDATA #REQUIRED
             total CDATA #REQUIRED
             result CDATA #REQUIRED>
<!ATTLIST ERROR type (external|internal|parser|tracker|xml|io) #REQUIRED
                module CDATA #IMPLIED
                start CDATA #IMPLIED>
<!ATTLIST ETM netmin CDATA #REQUIRED
              netavg CDATA #REQUIRED
              netmax CDATA #REQUIRED>
<!ATTLIST LC covered CDATA #REQUIRED
             total CDATA #REQUIRED
             result CDATA #REQUIRED>
<!ATTLIST MODULE name CDATA #REQUIRED>
<!ATTLIST MODULES tracked CDATA #REQUIRED
                  parsed CDATA #REQUIRED>
<!ATTLIST RFC rfc CDATA #REQUIRED>
<!ATTLIST SC covered CDATA #REQUIRED
             total CDATA #REQUIRED
             result CDATA #REQUIRED>
<!ATTLIST TEST date CDATA #REQUIRED
               version CDATA #IMPLIED
               time CDATA #REQUIRED>
<!ATTLIST WMC wmc CDATA #REQUIRED>
]>
```

- **<DC>** - the decision coverage metrics

  covered - the number of covered branches

  total - the number of all branches

  result - the fraction of (covered/total) in percent

- **<DOCSTRINGTEST>** - the output of the docstring tests
- **<ERROR>** - an error message

  type - the type of the error

  module - the name of the module, where the error occurred

  start - the test, represented by the start module, during which the error occurred

- **<ERRORS>** - the block of errors <ERROR>
- **<ETM>** - the execution time per method

  netmin - the minimum net execution time of all modules

  netavg - the average net execution time of all modules

  netmax - the maximum net execution time of all modules

- **<LC>** - the line coverage metrics

  covered - the number of covered lines

  total - the number of all covered lines

  result - the fraction of (covered/total) in percent

- **<METRICS>** - the block of metrics
- **<MODULE>** - the parsed module

name - the name of the parsed module

- **<MODULES>** - the block of parsed modules

  tracked - the number of all successfully tracked modules

  parsed - the number of all successfully tracked and parsed modules

- **<OUTPUT>** - the standard output of the system under test
- **<RFC>** the response for a class metrics

  rfc the metrics result value

- **<SC>** - the statement coverage metrics

  covered - te number of covered statements

  total - the number of all statements

  result - the fraction of (covered/total) in percent

- **<SIZE>** - the code size metrics

  result - the number of all extended AST nodes

- **<TEST>** - top block

  date - the date of performing the test

  version - the label of the test

  time - the time of performing the test

- **<UNITTEST>** - the output of the unit tests
- **<WMC>** - the weighted method per class

  wmc - the metrics result value

5.5 The Architecture of the new Unit Test Framework

The architecture of the new unit test framework Figure 11 on page 59 consists of five components. The main logic is coded in the `Unit_Test_FW_Tool.py`. The control graph on Figure 12 on page 60 depicts all important steps. The `Tracker.py` is responsible for the dynamic analysis and the `Parser.py` is responsible for generating the AST structures. The `Metrics.py` contains all metrics. The last component `Output.py` writes XML files.

First the options are parsed and evaluated so that all parameters are prepared for the next steps. The options like `help` or `version` terminate the framework already here.

In the second step, the global XML tree with specific nodes is prepared. Also the global XML file is opened if given. Otherwise the standard output is used. For details about XML tree, see the standard Python documentation about the `xml.dom` and the `xml.dom.minidom` modules

When the XML infrastructure is ready, we can enter the third step, where all given directories are recursively traversed and all Python modules containing unit tests or docstring tests are executed. In this step all paths and time measurements are recorded. See Section 5.5.1 on page 60 for details. If we want to test extremely huge system the record structures may become serious bottleneck.

The fourth step is responsible for preparing module XML tree and output file. In fact, it is similar to the second step with the difference that a slightly different XML tree is prepared and the module XML file is opened.

After having finished the dynamic analysis we can start to parse each tracked module. We generate an AST tree and extend it with other useful attributes to save the number of recursive traversals performed during the calculation of the metrics. See Section 5.5.2 on page 61 for details. Also multiple statement per line check is carried out here.

In the sixth step both results are available - the dynamic (execution paths, time measurements) and the static (AST) results. So we can calculate metrics for each module. Note, that the AST structures are released afterwards to save the memory.

Then these results are written to the module XML file. That happens in the seventh step.

The system level metrics are calculated in the eighth step by aggregating the module metrics results. We have to check in the global XML file that all modules to be evaluated have been processed successfully. The attributes `tracked` and `parsed` of the `<MODULES>` XML element in the global XML file must be equal to the number of all modules under test. If not then an error occurred and the results are not complete.

The last ninth step writes the system level results to the global XML file or to the standard output and terminates.

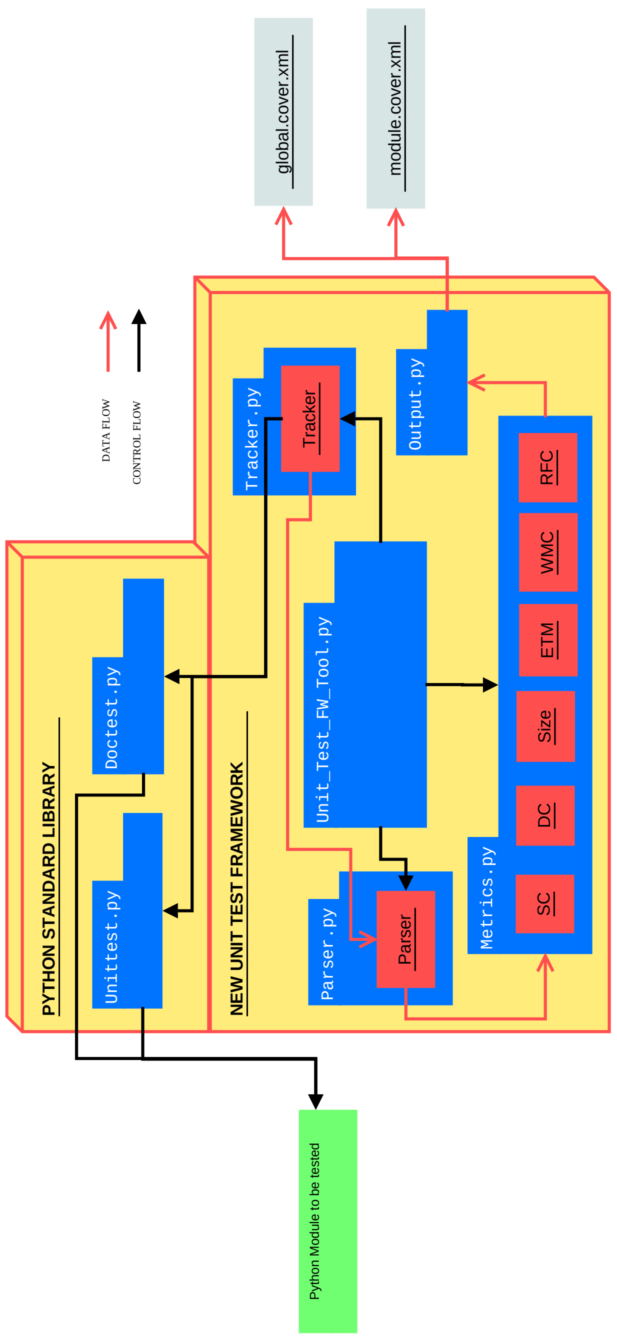

Figure 11: The New Unit Test Framework

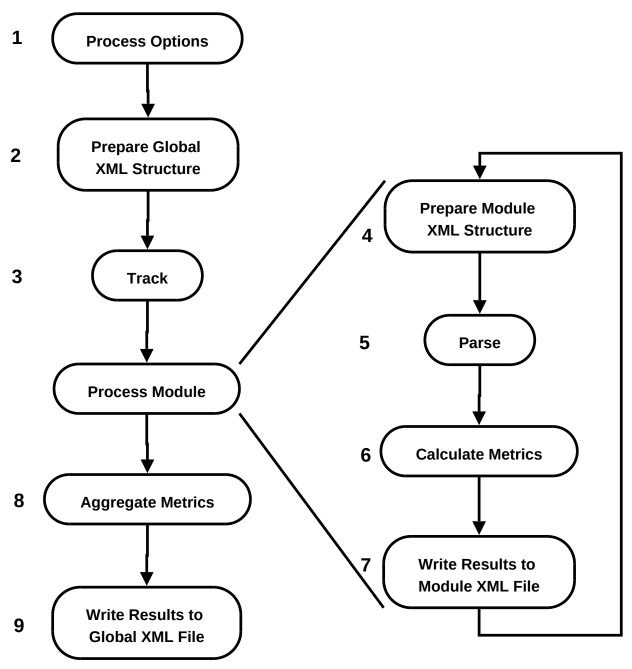
1
Process Options
2
Prepare Global XML Structure
3
Track
Process Module
8
Aggregate Metrics
9
Write Results to Global XML File
4
Prepare Module XML Structure
5
Parse
6
Calculate Metrics
7
Write Results to Module XML File

Figure 12: The Main Logic of the New Unit Test Framework

5.5.1 The Tracker Component

The dynamic analysis whose results are necessary for the statement and the decision coverage is performed by the tracking subsystem in the `Tracker.py`. The subsystem has two main functions; to record the paths and to measure execution time.

For recording the paths, we use the tracks data structure depicted in Figure 13 on the next page. This structure stores all independent paths for all modules. An *independent path* is an execution path in the module, which starts when the module is entered and it ends when the module is left. When the execution path enters another module meanwhile, this independent path is switched to another independent path belonging to that module. On the return from that module , the independent path of the interrupted module is restored for further recording. The recording is carried out by the `line` event callback function. To know, which independent path shall be restored, we store the indices from the “Files” and the “Independent Paths” arrays to a stack data structure. The independent path is represented by a “Line Numbers” array in the Table 13 on the facing page. After finishing the execution, the tracks data structure contains all independent paths for all modules.

For measuring the ETM we use the time data structure depicted in the Figure 14 on page 62. When a function is called the start gross time and the start net time are pushed to the stack. Both of these times are equal to the time instance when the function is entered. The *start gross time* and the *start net time* are at the beginning the instances of time when the function is entered. On the other hand, the *gross time* and the *net time* are intervals. See the Figure 10 on page 47. On the return from the function the gross time and the net time are calculated and stored in the time data structure. As we can see in the Figure 14 on the next page, each entry in the “Files” array is a tuple of containing the module name and the function name. Each of these entries points to another array of tuples consisting of the gross time and the net time. The length of this array represents in fact the number of calls of the function.

The calculation of the gross time is very simple. It is the difference between the start gross time and the end time. The *end time* is the time instance when the function is left. In case of the net time, the net time is the end time minus the start net time minus the sum of the gross times of all function’s subroutines. Therefore, we add these gross times to the function’s start net time, so that the difference between the end time and the start net time is the net time of the function.

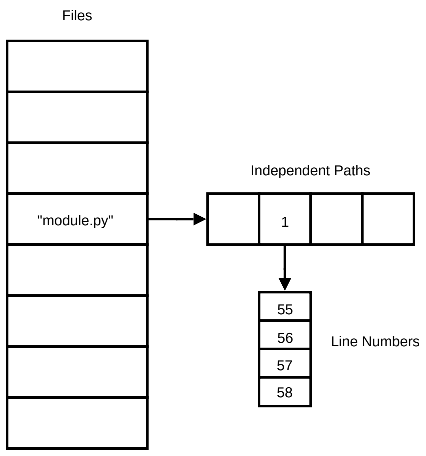

Figure 13: Tracks Data Structure

After the tracking subsystem finishes, every metrics gets the array of the independent paths as the result of the dynamic analysis.

5.5.2 The Parser Component

The static analysis, which generates the extended AST is performed by the `Parser.py`. We want to introduce the extensions only, since the AST is documented in the Python standard documentation. As mentioned earlier, the extensions shall save the number of recursive traversals and are in fact additional attributes of the extended AST nodes. They are:

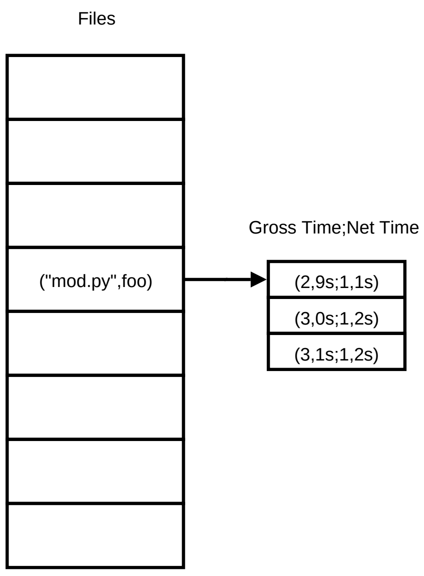

Figure 14: Time Data Structure

- *id* - every node has unique id, which is useful when metrics results are manually checked.
- *count* - stores, how many times the node has been executed.
- *end* - if true, it warns that the control flow has ended here. This is needed for decision coverage when detecting branches.
- *next* - the next node concerning the source code
- *previous* - the previous node concerning the source code
- *parent* - the parent node concerning AST. This attribute improves the navigation in the AST.
- *successors* - the successing nodes concerning execution
- *predecessors* - the preceding nodes concerning execution

When the AST is generated and extended with these additional attributes, it is given to all metrics.

5.6 Summary

This chapter described the requirements and limitations of the new unit test framework. The requirements were the implementation of the maximum number of metrics and the possibility of measuring the execution time. To keep the complexity of the metrics calculations low, the metrics used the syntactic level of the measured source code only. Another limitation was the line resolution of the tracking system, which is one of two cornerstones on which all metrics are based on. The second cornerstone is the abstract syntax tree (AST), which is received as a result of the parsing system.

With regard to the requirements and limitations we chose the following metrics that could be implemented: the SC, the DC, the SIZE, the ETM, the WMC, the RFC, and the LC. Two other metrics, the DIT and the NOC, which also met the requirements and the limitations, were not implemented. It was decided to implement them later.

This chapter also dealt with the interface of the new unit test framework. The framework accepted all Python modules, especially those which implemented the unit tests and the docstring tests. The result was written into the module XML files and the global XML file.

Finally, the architecture of the new unit test framework was described. All given Python modules were executed and tracked to extract the execution path. Then, the metrics were calculated for each module parsed, with the metrics results having been written into XML files.

6 Evaluation

After designing and implementing the new unit test framework, we analyze the results generated by the framework. The generated results are shown in the following tables: Table 5 on page 69, Table 6.1 on page 71 and Table 6.2 on page 84.

Table 5 on page 69 is an ancillary table containing the long names of the start modules. Note: A start module is an entry point of the system under test. A unique number (shown in the left column of Table 5 on page 69) is assigned to every start module. This number `No.` is then used in the other two tables (Table 6.1 on page 71 and Table 6.2 on page 84) as a reference number.

Before we analyze the metrics and the framework-internal results, we describe what has been tested in which way. The first question can be answered quite simply. We took all the Python modules beginning with the `Test` prefix as the start modules, because the company convention requires exactly these modules to contain all test case implementations.

As far as the way of testing is concerned, the new unit test framework supports two ways of testing:

1. We provide a list of start modules that are evaluated at once.
2. We start the framework evaluation per start module.

The advantage of the first alternative is that it yields results combining all per metrics. For example, if two test case implementations of two different start modules test something in a third module, then the SC of the third module is a combination of two SCs produced by these two test case implementations.

The advantage of the second alternative is that we have metrics results per start module. Table 6.1 on page 71 and Table 6.2 on page 84 show the metrics results and internal statistical data. We took only the second testing method, because we are interested in results per start module.

No.	**Start Modules**
0	Testoize.py
1	Testoize_pkg.py
2	_BETH/Test_Action.py
3	_BETH/Test_Case.py
4	_BETH/Test_Case_Template.py
5	_BETH/Test_Command.py

No.	**Start Modules**
6	_BETH/Test_Interface.py
7	_BETH/Test_Result_File.py
8	_BETH/Test_Script.py
9	_CDT/_Sched/Test_FA.py
10	_CPT/_U_Test/Test_UI_Spec_Parser.py
11	_DLT/_U_Test/Test_BBlock_Lists.py
12	_DLT/_U_Test/Test_Byte_Order.py
13	_DLT/_U_Test/Test_DLT_COM_Buffer.py
14	_DLT/_U_Test/Test_DLT_COM_Frame.py
15	_DLT/_U_Test/Test_DLT_COM_GDL.py
16	_DLT/_U_Test/Test_Descriptor_Blocks.py
17	_DLT/_U_Test/Test_File_Loader.py
18	_DLT/_U_Test/Test_Node_Config.py
19	_ETA/_U_Test/Test_Msg_Type_P_uses_Enum_Type.py
20	_ETT/_TD/_U_Test/Test_Measurement.py
21	_HDT/_EXC/_BAT/_U_Test/Test_Bit_Assignment_Worksheet.py
22	_HDT/_EXC/_DMS/_U_Test/Test_Dms_Reader.py
23	_HDT/_EXC/_U_Test/Test_Cell_Range.py
24	_HDT/_EXC/_U_Test/Test_Data_Reader.py
25	_HDT/_EXC/_U_Test/Test_Excel_Handler.py
26	_HDT/_EXC/_U_Test/Test_TMC_Data_Reader.py
27	_HDT/_EXC/_U_Test/Test_TMC_Work_Sheet.py
28	_HDT/_LGO/_U_Test/Test_Round.py
29	_HDT/_Sched/_U_Test/Test_Frequency_Selector.py
30	_HDT/_Sched/_U_Test/Test_Schedule_Table.py
31	_HDT/_Sched/_U_Test/Test_Scheduler.py
32	_LIN/_FRAME/_U_Test/Test_Diagnostic_Frame.py
33	_LIN/_FRAME/_U_Test/Test_ET_Frame.py
34	_LIN/_FRAME/_U_Test/Test_Frame_A.py
35	_LIN/_FRAME/_U_Test/Test_Frame_Mixin.py
36	_LIN/_FRAME/_U_Test/Test_Frame_carries_Unconditional_Frame.py
37	_LIN/_FRAME/_U_Test/Test_Master_Request_Frame.py
38	_LIN/_FRAME/_U_Test/Test_Slave_Response_Frame.py
39	_LIN/_FRAME/_U_Test/Test_Sporadic_Frame.py
40	_LIN/_FRAME/_U_Test/Test_Sporadic_Frame_carries_Unconditional_Frame.py
41	_LIN/_FRAME/_U_Test/Test_Unconditional_Frame.py

No.	Start Modules
42	_LIN/_SIG/_U_Test/Test_Byte_Array.py
43	_LIN/_SIG/_U_Test/Test_Data_Signal.py
44	_LIN/_SIG/_U_Test/Test_Data_Signal_depends_on_Data_Signal.py
45	_LIN/_SIG/_U_Test/Test_Diagnostic_Signal.py
46	_LIN/_SIG/_U_Test/Test_Msg_Type_ASCII.py
47	_LIN/_SIG/_U_Test/Test_Msg_Type_BCD.py
48	_LIN/_SIG/_U_Test/Test_Msg_Type_L.py
49	_LIN/_SIG/_U_Test/Test_Msg_Type_Phy.py
50	_LIN/_SIG/_U_Test/Test_Scalar_Signal.py
51	_LIN/_SIG/_U_Test/Test_Scalar_Signal_uses_Signal_Encoding.py
52	_LIN/_SIG/_U_Test/Test_Signal.py
53	_LIN/_SIG/_U_Test/Test_Signal_Encoding.py
54	_LIN/_SIG/_U_Test/Test_Signal_Encoding_uses_Msg_Type.py
55	_LIN/_U_Test/Test_Centurion.py
56	_LIN/_U_Test/Test_Cohort.py
57	_LIN/_U_Test/Test_Cohort_Mode.py
58	_LIN/_U_Test/Test_Cohort_Mode_sends_Frame_A_in_Round_A.py
59	_LIN/_U_Test/Test_Cohort_Mode_uses_Data_Signal.py
60	_LIN/_U_Test/Test_Cohort_Mode_uses_Frame_A.py
61	_LIN/_U_Test/Test_Cohort_Mode_uses_Special_Frame.py
62	_LIN/_U_Test/Test_Configuration.py
63	_LIN/_U_Test/Test_Data_Signal_in_Unconditional_Frame.py
64	_LIN/_U_Test/Test_Diagnostic_Signal_in_Diagnostic_Frame.py
65	_LIN/_U_Test/Test_Legionary.py
66	_LIN/_U_Test/Test_Lin_Node_provides_Data_Signal.py
67	_LIN/_U_Test/Test_Lin_Node_requires_Data_Signal.py
68	_LIN/_U_Test/Test_Lin_Node_uses_Data_Signal.py
69	_LIN/_U_Test/Test_Node_A.py
70	_LIN/_U_Test/Test_Node_A_in_Cohort.py
71	_MGW/_U_Test/Test_Monitor_Gateway.py
72	_NDT/_Sched2/_Tester/Test_Driver.py
73	_TFL/_Meta/Test.py
74	_TFL/_Meta/Test_2.py
75	_TFL/_Meta/Test_3.py
76	_TGW/_U_Test/Test_Slideshow.py
77	_TGW/_U_Test/Test_Slider_Entry.py

No.	Start Modules
78	_TGW/_U_Test/Test_Add_on_Feedback.py
79	_TGW/_U_Test/Test_Tooltips.py
80	_TGW/_U_Test/Test_Add_on_File_Dialog.py
81	_TGW/_U_Test/Test_Toolbar.py
82	_TGW/_U_Test/Test_Add_on_History.py
83	_TGW/_U_Test/Test_Toplevel.py
84	_TGW/_U_Test/Test_Add_on_Led.py
85	_TGW/_U_Test/Test_Treeview.py
86	_TGW/_U_Test/Test_Add_on_Text_View.py
87	_TGW/_U_Test/Test_Window.py
88	_TGW/_U_Test/Test_Alignment.py
89	_TGW/_U_Test/Test_Box.py
90	_TGW/_U_Test/Test_Button.py
91	_TGW/_U_Test/Test_Button_Box.py
92	_TGW/_U_Test/Test_Check_Button.py
93	_TGW/_U_Test/Test_Clipboard.py
94	_TGW/_U_Test/Test_Combo_Box.py
95	_TGW/_U_Test/Test_Combo_Box_Entry.py
96	_TGW/_U_Test/Test_Composite_Entry.py
97	_TGW/_U_Test/Test_Cursor.py
98	_TGW/_U_Test/Test_Data_Model.py
99	_TGW/_U_Test/Test_Dialog.py
101	_TGW/_U_Test/Test_Entry.py
102	_TGW/_U_Test/Test_Event_Box.py
103	_TGW/_U_Test/Test_Expander.py
104	_TGW/_U_Test/Test_File_Dialog.py
105	_TGW/_U_Test/Test_Frame.py
106	_TGW/_U_Test/Test_Gtk_Object.py
107	_TGW/_U_Test/Test_Handle_Box.py
108	_TGW/_U_Test/Test_History.py
109	_TGW/_U_Test/Test_History_Manager.py
110	_TGW/_U_Test/Test_Icon_Pool.py
111	_TGW/_U_Test/Test_Image.py
112	_TGW/_U_Test/Test_Image_Manager.py
113	_TGW/_U_Test/Test_Interpreter_Entry.py
114	_TGW/_U_Test/Test_Label.py

No.	Start Modules
115	_TGW/_U_Test/Test_Label_Separator.py
116	_TGW/_U_Test/Test_Layout.py
117	_TGW/_U_Test/Test_List.py
118	_TGW/_U_Test/Test_List_Entry.py
119	_TGW/_U_Test/Test_Menu.py
121	_TGW/_U_Test/Test_Message_Window.py
122	_TGW/_U_Test/Test_Notebook.py
123	_TGW/_U_Test/Test_Pane.py
124	_TGW/_U_Test/Test_Popup_Window.py
125	_TGW/_U_Test/Test_Progress_Bar.py
126	_TGW/_U_Test/Test_Progress_Window.py
127	_TGW/_U_Test/Test_Radio_Button.py
128	_TGW/_U_Test/Test_Radio_Group.py
129	_TGW/_U_Test/Test_Screen.py
130	_TGW/_U_Test/Test_Scroll_Bar.py
131	_TGW/_U_Test/Test_Scrolled_Window.py
132	_TGW/_U_Test/Test_Separator.py
133	_TGW/_U_Test/Test_Sheet.py
134	_TGW/_U_Test/Test_Signal.py
135	_TGW/_U_Test/Test_Size_Group.py
136	_TGW/_U_Test/Test_Slider.py
137	_TGW/_U_Test/Test_Spinner.py
138	_TGW/_U_Test/Test_Statusbar.py
139	_TGW/_U_Test/Test_T_Box.py
140	_TGW/_U_Test/Test_Table.py
141	_TGW/_U_Test/Test_Text_Buffer.py
142	_TGW/_U_Test/Test_Text_View.py
143	_TGW/_U_Test/Test_Text_Window.py
144	_TGW/_U_Test/Test_Toggle_Button.py
145	_TIM/_U_Test/Test_Config_Data.py
146	_TIM/_U_Test/Test_File.py
147	_TIM/_U_Test/Test_Installer.py
148	_TIM/_U_Test/Test_Manifest.py
149	_TIM/_U_Test/Test_Persistent_Info.py
150	_TIM/_U_Test/Test_Plugin.py
151	_TTA/_FBX/_U_Test/Test_XML_Exporter.py

No.	Start Modules
152	_TTA/_FBX/_U_Test/Test_XML_Importer.py
153	_TTA/_FBX/_U_Test/Test_XML_Structure.py
154	_TTA/_FBX/_U_Test/Test_XML_Structure_With_ID.py
155	_TTA/_FBX/_U_Test/Test_XML_Structure_With_IDREF.py
156	_TTA/_FTC/_TDCOM/_U_Test/Test_Frame.py
157	_TTA/_FTC/_TDCOM/_U_Test/Test_Interrupt_Schedule.py
158	_TTA/_FTC/_TDCOM/_U_Test/Test_Interrupt_Scheduler.py
159	_TTA/_FTC/_TDCOM/_U_Test/Test_Table_Checker.py
160	_XCD/_Sched/Test_FA.py
161	_XSS/_GEN/_U_Test/Test_Parameter_Object.py

Table 5: Start Modules

6.1 Metrics Evaluation

Table 6.1 on page 71 shows all metrics results for all used start modules found. The column headers have the following meanings.

No. The reference number of the respective start module. See Table 5.

Tracked The number of modules tracked. It is a set of all modules successfully executed during test. In other words, **Tracked** is the number of modules successfully passing step 3, which is shown in Figure 12 on page 60.

Parsed The number of modules parsed, i.e., the number of modules successfully passing steps 4, 5, 6, 7, which is shown in Figure 12 on page 60.

Lines The total number of lines of all the modules parsed.

Errors The number of modules containing at least one error message.

Warnings The number of modules containing at least one warning. There is only one warning which says that the coding guidelines have been violated by placing more than one statement in a line.

LC The total line coverage of all modules. See Section 5.3.3 on page 47.

SC The total statement coverage of all modules. See Section 5.3.1 on page 44.

DC The total decision coverage of all modules. See Section 5.3.2 on page 45.

Size The total code size of all modules. The unit is AST node. See Section 5.3.3 on page 47.

WMC The total weighted method per class of all modules. See Section 5.3.5 on page 47.

RFC The total response for a class of all modules. See See Section 5.3.6 on page 48.

ETM min The sum of the minimum execution times per method of all modules. See Section 5.3.4 on page 47.

ETM avg The sum of the average execution times per method of all modules. See Section 5.3.4 on page 47.

ETM max The sum of the maximum execution times per method of all modules. See Section 5.3.4 on page 47.

No.	Modules					Metrics								
	Tracked	**Parsed**	**Lines**	**Errors**	**Warnings**	**LC [%]**	**SC [%]**	**DC [%]**	**Size [Node]**	**WMC [wmc]**	**RFC [rfc]**	**ETM min [s]**	**ETM avg [s]**	**ETM max [s]**
0	22	22	2183	0	0	26	26	8	10381	622	686	0.19	0.21	0.38
1	20	20	2105	0	0	26	27	8	10021	622	686	0.19	0.21	0.36
2	18	18	2343	0	3	26	26	6	11460	653	659	0.20	0.21	0.30
3	28	28	3829	1	7	20	20	5	18882	1260	1260	0.20	0.23	0.35
4	23	23	2929	0	4	25	25	7	14395	854	949	0.19	0.21	0.34
5	14	14	1210	0	1	28	28	9	5590	228	216	0.11	0.12	0.18
6	7	7	572	1	1	12	13	4	2562	191	191	0.02	0.02	0.02
7	29	29	3997	1	8	20	19	4	19737	1366	1362	0.16	0.19	0.33
8	6	6	537	0	1	19	20	9	2488	178	188	0.01	0.01	0.02
9	22	22	2380	0	2	29	30	9	11195	715	934	0.17	0.21	0.41
10	82	82	7256	1	0	27	28	7	32004	2435	2799	0.54	0.59	0.95
11	43	43	3801	0	0	29	31	12	18469	1223	1347	0.27	0.28	0.70
12	15	15	1271	0	0	37	37	18	6162	377	467	0.07	0.08	0.28
13	11	11	788	0	0	39	41	25	3743	275	383	0.06	0.07	0.21
14	10	10	701	0	0	32	34	22	3259	225	325	0.03	0.03	0.08
15	14	14	1406	0	0	64	60	38	7518	657	763	0.20	0.19	0.50

No.	Modules					Metrics								
	Tracked	**Parsed**	**Lines**	**Errors**	**Warnings**	**LC [%]**	**SC [%]**	**DC [%]**	**Size [Node]**	**WMC [wmc]**	**RFC [rfc]**	**ETM min [s]**	**ETM avg [s]**	**ETM max [s]**
16	48	48	4026	0	0	29	29	12	18638	1133	1357	0.20	0.21	0.56
17	28	28	2623	0	0	29	31	12	13073	688	738	0.19	0.22	0.42
18	5	5	394	1	0	20	21	7	1834	76	95	0.01	0.01	0.01
19	1	1	19	1	0	5	5	0	83	1	8	0.00	0.00	0.00
20	20	20	1403	0	0	46	46	25	6445	507	602	0.17	0.15	0.45
21	43	43	2736	1	0	33	34	12	11913	834	930	0.22	0.28	0.48
22	5	5	387	1	0	6	7	2	1789	154	165	0.01	0.01	0.01
23	1	1	18	1	0	5	5	0	79	2	6	0.00	0.00	0.00
24	1	1	18	1	0	5	5	0	74	1	8	0.00	0.00	0.00
25	1	1	55	1	0	1	1	0	272	2	34	0.01	0.01	0.01
26	1	1	127	1	0	0	0	0	597	8	97	0.00	0.00	0.00
27	1	1	22	1	0	4	4	0	82	1	9	0.01	0.01	0.01
28	1	1	145	1	0	0	0	0	703	19	86	0.00	0.00	0.00
29	1	1	206	1	0	0	0	0	1165	121	103	0.00	0.00	0.00
30	1	1	67	1	0	1	1	0	307	6	40	0.01	0.01	0.01
31	1	1	181	1	0	0	0	0	1305	64	108	0.00	0.00	0.00

No.	Modules					Metrics								
	Tracked	Parsed	Lines	Errors	Warnings	LC [%]	SC [%]	DC [%]	Size [Node]	WMC [wmc]	RFC [rfc]	ETM min [s]	ETM avg [s]	ETM max [s]
32	1	1	24	1	0	4	5	0	94	2	10	0.00	0.00	0.00
33	1	1	99	1	0	1	1	0	542	7	72	0.00	0.00	0.00
34	1	1	30	1	0	3	4	0	113	3	11	0.00	0.00	0.00
35	1	1	93	1	0	1	1	0	430	4	75	0.00	0.00	0.00
36	1	1	32	1	0	3	3	0	161	1	20	0.00	0.00	0.00
37	1	1	20	1	0	5	5	0	93	2	11	0.00	0.00	0.00
38	1	1	14	1	0	7	7	0	54	2	5	0.00	0.00	0.00
39	1	1	34	1	0	2	3	0	159	3	21	0.00	0.00	0.00
40	1	1	34	1	0	2	3	0	161	2	20	0.01	0.01	0.01
41	1	1	124	1	0	0	1	0	580	8	77	0.00	0.00	0.00
42	1	1	50	1	0	2	3	0	203	6	19	0.00	0.00	0.00
43	2	2	160	1	0	20	29	15	596	24	61	0.00	0.00	0.00
44	1	1	68	1	0	1	3	0	225	7	23	0.00	0.00	0.00
45	1	1	14	1	0	7	7	0	55	2	5	0.01	0.01	0.01
46	1	1	14	1	0	7	7	0	53	1	4	0.00	0.00	0.00
47	1	1	14	1	0	7	7	0	53	1	4	0.00	0.00	0.00

No.	Modules					Metrics									
	Tracked	Parsed	Lines	Errors	Warnings	LC [%]	SC [%]	DC [%]	Size [Node]	WMC [wmc]	RFC [rfc]	ETM min [s]	ETM avg [s]	ETM max [s]
48	1	1	20	1	0	5	5	0	78	3	7	0.00	0.00	0.00
49	1	1	38	1	0	2	4	0	122	3	12	0.00	0.00	0.00
50	1	1	180	1	0	0	1	0	702	11	90	0.00	0.00	0.00
51	1	1	21	1	0	4	5	0	86	1	8	0.00	0.00	0.00
52	2	2	96	1	0	34	36	15	393	22	38	0.01	0.01	0.01
53	1	1	53	1	0	1	2	0	252	6	36	0.00	0.00	0.00
54	2	2	92	1	0	35	38	15	368	21	36	0.01	0.01	0.01
55	1	1	36	1	0	2	3	0	147	4	17	0.00	0.00	0.00
56	1	1	45	1	0	2	2	0	209	6	26	0.00	0.00	0.00
57	1	1	110	1	0	0	1	0	587	11	88	0.00	0.00	0.00
58	1	1	40	1	0	2	2	0	194	2	27	0.00	0.00	0.00
59	1	1	45	1	0	2	3	0	167	3	21	0.00	0.00	0.00
60	1	1	49	1	0	2	2	0	203	4	27	0.01	0.01	0.01
61	1	1	43	1	0	2	3	0	167	3	21	0.00	0.00	0.00
62	1	1	49	1	0	2	2	0	264	2	39	0.00	0.00	0.00
63	1	1	182	1	0	0	0	0	884	20	119	0.00	0.00	0.00

No.	Modules					Metrics								
	Tracked	Parsed	Lines	Errors	Warnings	LC [%]	SC [%]	DC [%]	Size [Node]	WMC [wmc]	RFC [rfc]	ETM min [s]	ETM avg [s]	ETM max [s]
64	1	1	47	1	0	2	2	0	231	5	27	0.00	0.00	0.00
65	1	1	119	1	0	0	1	0	451	15	52	0.00	0.00	0.00
66	1	1	33	1	0	3	3	0	166	3	21	0.00	0.00	0.00
67	1	1	21	1	0	4	5	0	85	1	8	0.00	0.00	0.00
68	1	1	13	1	0	7	9	0	42	1	2	0.00	0.00	0.00
69	1	1	56	1	0	1	1	0	309	4	43	0.00	0.00	0.00
70	1	1	17	1	0	5	5	0	81	1	7	0.00	0.00	0.00
71	78	78	5718	0	1	28	31	8	26386	1819	2015	0.37	0.38	0.67
72	16	16	1156	0	0	25	26	11	5300	259	314	0.02	0.03	0.11
73	1	1	106	0	0	93	93	0	392	20	47	0.01	0.00	0.02
74	1	1	30	1	0	80	80	75	104	16	10	0.00	0.00	0.01
75	1	1	27	1	0	74	74	0	116	8	20	0.00	0.00	0.00
76	25	19	2377	6	0	33	27	9	9984	632	718	0.13	0.14	0.27
77	25	19	2346	6	0	33	27	9	9853	628	695	0.11	0.13	0.28
78	25	19	2344	6	0	33	27	9	9832	628	693	0.13	0.15	0.28
79	25	19	2350	6	0	33	27	9	9872	626	701	0.14	0.15	0.27

No.	Modules					Metrics								
	Tracked	**Parsed**	**Lines**	**Errors**	**Warnings**	**LC [%]**	**SC [%]**	**DC [%]**	**Size [Node]**	**WMC [wmc]**	**RFC [rfc]**	**ETM min [s]**	**ETM avg [s]**	**ETM max [s]**
80	25	19	2340	6	0	34	27	9	9818	626	692	0.15	0.17	0.29
81	25	19	2401	6	0	33	27	9	10031	634	719	0.12	0.13	0.25
82	25	19	2354	6	0	33	27	9	9882	626	701	0.13	0.14	0.27
83	25	19	2347	6	0	33	27	9	9841	626	697	0.15	0.17	0.31
84	25	19	2340	6	0	34	27	9	9819	626	692	0.17	0.16	0.31
85	24	18	2387	6	0	32	25	8	10348	621	742	0.15	0.16	0.28
86	25	19	2340	6	0	34	27	9	9818	626	692	0.14	0.15	0.27
87	24	18	2312	6	0	33	26	8	9702	619	686	0.18	0.19	0.31
88	25	19	2344	6	0	33	27	9	9840	627	696	0.13	0.13	0.27
89	25	19	2358	6	0	33	27	9	9910	632	703	0.19	0.19	0.33
90	25	19	2388	6	0	33	26	9	10085	637	734	0.12	0.13	0.23
91	25	19	2375	6	0	33	27	9	9984	633	717	0.16	0.16	0.33
92	25	19	2370	6	0	33	27	9	9994	636	717	0.17	0.19	0.31
93	24	18	2271	6	0	33	26	8	9544	609	665	0.14	0.15	0.29
94	25	19	2353	6	0	33	27	9	9881	630	700	0.16	0.18	0.32
95	25	19	2374	6	0	33	27	9	10010	634	715	0.14	0.15	0.31

No.	Modules					Metrics								
	Tracked	**Parsed**	**Lines**	**Errors**	**Warnings**	**LC [%]**	**SC [%]**	**DC [%]**	**Size [Node]**	**WMC [wmc]**	**RFC [rfc]**	**ETM min [s]**	**ETM avg [s]**	**ETM max [s]**
96	25	19	2359	6	0	33	27	9	9891	630	700	0.10	0.12	0.24
97	26	20	2434	6	0	32	26	8	10232	677	734	0.15	0.15	0.26
98	25	19	2368	6	0	33	27	9	10017	628	710	0.15	0.14	0.25
99	25	19	2351	6	0	33	27	9	9855	627	696	0.14	0.14	0.30
101	25	19	2359	6	0	33	27	9	9876	628	700	0.12	0.12	0.25
102	25	19	2346	6	0	33	27	9	9834	627	695	0.14	0.14	0.25
103	25	19	2353	6	0	33	27	9	9869	627	700	0.12	0.13	0.28
104	25	19	2365	6	0	33	27	9	9956	634	717	0.19	0.19	0.31
105	25	19	2373	6	0	33	27	9	9989	631	713	0.15	0.16	0.28
106	25	19	2356	6	0	33	27	9	9920	628	706	0.13	0.14	0.27
107	25	19	2343	6	0	33	27	9	9826	626	695	0.13	0.14	0.27
108	25	19	2349	6	0	33	27	9	9844	627	696	0.16	0.15	0.29
109	25	19	2355	6	0	33	27	9	9866	626	697	0.16	0.17	0.31
110	25	19	2363	6	0	33	27	9	9963	629	710	0.12	0.14	0.27
111	25	19	2340	6	0	34	27	9	9811	626	691	0.13	0.13	0.27
112	25	19	2342	6	0	33	27	9	9820	626	691	0.16	0.16	0.31

No.	Modules					Metrics								
	Tracked	**Parsed**	**Lines**	**Errors**	**Warnings**	**LC [%]**	**SC [%]**	**DC [%]**	**Size [Node]**	**WMC [wmc]**	**RFC [rfc]**	**ETM min [s]**	**ETM avg [s]**	**ETM max [s]**
113	25	19	2343	6	0	33	27	9	9826	626	696	0.14	0.15	0.24
114	25	19	2347	6	0	33	27	9	9853	627	697	0.14	0.14	0.29
115	25	19	2341	6	0	34	27	9	9826	626	695	0.14	0.15	0.28
116	25	19	2366	6	0	33	27	9	9974	635	713	0.15	0.16	0.28
117	25	19	2370	6	0	33	27	9	9946	628	714	0.22	0.23	0.34
118	25	19	2343	6	0	33	27	9	9828	626	695	0.14	0.13	0.23
119	24	18	2435	6	0	31	24	8	10413	618	797	0.15	0.15	0.24
121	25	19	2358	6	0	33	27	9	9922	633	707	0.15	0.15	0.31
122	25	19	2411	6	0	33	26	9	10265	634	771	0.15	0.17	0.30
123	25	19	2355	6	0	33	27	9	9902	629	705	0.13	0.14	0.25
124	25	19	2355	6	0	33	27	9	9873	627	701	0.16	0.16	0.29
125	25	19	2349	6	0	33	27	9	9841	626	693	0.14	0.14	0.24
126	25	19	2350	6	0	33	27	9	9850	627	696	0.13	0.14	0.27
127	25	19	2358	6	0	33	27	9	9911	627	706	0.14	0.16	0.31
128	25	19	2359	6	0	33	27	9	9911	630	705	0.16	0.17	0.32
129	25	19	2342	7	0	33	27	9	9824	626	694	0.14	0.15	0.25

No.	Modules					Metrics								
	Tracked	Parsed	Lines	Errors	Warnings	LC [%]	SC [%]	DC [%]	Size [Node]	WMC [wmc]	RFC [rfc]	ETM min [s]	ETM avg [s]	ETM max [s]
130	25	19	2338	6	0	34	27	9	9813	626	691	0.16	0.17	0.28
131	25	19	2359	6	0	33	27	9	9916	634	705	0.13	0.16	0.26
132	25	19	2338	6	0	34	27	9	9810	626	692	0.14	0.16	0.29
133	25	19	2405	7	0	33	26	9	10096	634	722	0.17	0.18	0.32
134	25	19	2359	6	0	33	27	9	9929	632	706	0.14	0.16	0.30
135	24	18	2304	6	0	33	26	8	9733	617	695	0.13	0.15	0.28
136	25	19	2359	6	0	33	27	9	9925	630	707	0.17	0.16	0.28
137	25	19	2351	6	0	33	27	9	9863	628	698	0.13	0.14	0.30
138	25	19	2363	6	0	33	27	9	9937	628	714	0.10	0.11	0.27
139	25	19	2375	6	0	33	27	9	10033	637	722	0.18	0.18	0.31
140	25	19	2396	6	0	33	26	9	10186	654	735	0.19	0.18	0.29
141	24	18	2400	6	0	31	25	8	10493	667	791	0.11	0.12	0.24
142	24	18	2311	6	0	33	26	8	9788	611	704	0.14	0.16	0.27
143	25	19	2340	6	0	34	27	9	9813	628	691	0.14	0.15	0.27
144	25	19	2357	6	0	33	27	9	9923	628	708	0.15	0.15	0.27
145	16	16	1472	0	0	29	29	12	6600	497	516	0.17	0.18	0.28

No.	Modules					Metrics								
	Tracked	**Parsed**	**Lines**	**Errors**	**Warnings**	**LC [%]**	**SC [%]**	**DC [%]**	**Size [Node]**	**WMC [wmc]**	**RFC [rfc]**	**ETM min [s]**	**ETM avg [s]**	**ETM max [s]**
146	17	17	1249	0	0	29	30	13	5719	387	469	0.15	0.15	0.21
147	45	45	6100	1	0	40	41	6	29968	1573	1756	0.58	0.62	0.87
148	16	16	1504	0	0	26	25	10	6960	498	611	0.15	0.15	0.22
149	14	14	1341	0	0	40	42	26	6194	425	507	0.17	0.20	0.35
150	17	17	1924	0	0	27	28	12	8626	682	703	0.16	0.16	0.26
151	1	1	84	1	0	2	2	0	349	17	34	0.00	0.00	0.00
152	1	1	149	1	0	1	1	0	703	21	83	0.00	0.00	0.00
153	1	1	103	1	0	1	2	0	435	22	48	0.00	0.00	0.00
154	1	1	28	1	0	7	8	0	117	6	13	0.00	0.00	0.00
155	1	1	24	1	0	8	9	0	93	3	9	0.01	0.01	0.01
156	55	55	5681	0	0	22	23	6	25104	1958	2152	0.24	0.28	0.61
157	55	55	5641	0	0	23	24	7	25038	1940	2140	0.21	0.28	0.64
158	55	55	5709	0	0	27	27	11	25356	1996	2137	0.31	0.39	0.96
159	40	40	3313	0	0	24	26	14	14828	1109	1225	0.17	0.26	0.78
160	20	20	2358	0	2	29	29	9	11115	711	925	0.21	0.23	0.44
161	14	14	934	0	0	52	52	30	4413	301	452	0.23	0.21	0.44

No.	Modules					Metrics								
	Tracked	Parsed	Lines	Errors	Warnings	LC [%]	SC [%]	DC [%]	Size [Node]	WMC [wmc]	RFC [rfc]	ETM min [s]	ETM avg [s]	ETM max [s]

Table 6: Metrics Evaluation

The most important piece of information listed in Table 6.1 on page 71 is the results of the statement coverage metrics, the decision coverage metrics and the less important line coverage metrics, which serves for comparison. According to Table 6.1 on page 71, the SC averages 21,1%, the DC averagess 7,2% and the LC averages 23,3%. These values are not surprising, because they were expected by *TTTech*'s developers. The WMC metrics and the RFC metrics are newly introduced metrics, thus we cannot score their values. We can only watch their behavior in relation to the size of the modules parsed. The last metrics, ETM, is remarkable, because the sum of the `ETM max` is only 24 seconds, although the entire test took more than 3 hours. This large discrepancy was caused by the instrumentation of the unit test framework code.

To correctly read Table 6.1 on page 71, we have to focus on the `Errors` and `Warnings` columns. Note that the `Errors` column states how many modules failed during test and not how many errors occurred. The LC, SC and DC metrics might be strongly affected when the respective columns are non-zero, because the test did not finish.

The warning column states the number of modules where the TTTech coding guidelines were violated. As is the case with the Errors column, the warning column does not state how many times the coding guidelines were violated. The non-zero value can influence the LC, the SC and the DC metrics. Details about this problem are presented in Section 5.2.1 on page 42.

We also want to know the origin of the warnings and errors that make the results incorrect. In the event of the warnings, there is only one cause, namely the violated coding guidelines. According to the coding guidelines, no body statement may follow after `if`, `for`, `while` and `except` conditions (that is, after "`:`").

The errors that occurred during the tests were mostly import module errors. As the appropriate environment is not saved in the repository, its preparation is out of the scope of this thesis. Furthermore, there were a few other errors where the unit tests failed.

If errors occur during a test, the values of the metrics can be surprising. Two such cases are listed in Table 6.1 on page 71. An example of the first case is test No. 65. The SC is non-zero, although the LC is zero. This might happen due to rounding the results down to integers. In this example, the LC = $1/119$ (covered / total) = 0%, and the SC = $1/62$ (covered / total) = 1%. The number of lines is higher than the number of statements, beause lots of statements extend over multiple lines.

The second case is represented by test No. 75. The SC=$20/27$ (covered / total) = 74%, whereas the DC = $0/0$ (covered / total) is interpreted as 0%. At first glance, it is impossible that the DC is 0% when the SC is so high. But, examining this case, we can see that this scenario, with no decision covered within 27 statements, is probable. The value 0% of the DC means either no decision was covered or there are no decision in code at all.

6.2 Internal Data Structures Evaluation

This section gives an overview of performance data listed in Table 6.2 on the next page. The columns are described below.

Max. Depth The maximum size of the internal stack reached during execution. See Section 5.5.1 on page 60.

Shortest The number of nodes in the shortest path tracked in the tracks data structure, shown in Figure 13 on page 61.

Average The average number of nodes in the paths in the tracks data structure, shown in Figure 13 on page 61.

Longest The number of nodes in the longest path tracked in the tracks data structure, shown in Figure 13 on page 61.

Number The number of all the individual paths tracked.

Min. The smallest AST generated from all the modules parsed.

Avg. The average number of nodes per AST generated from all the modules parsed.

Max. The largest AST generated from all the modules parsed.

Measure. The number of all time measurements of all method calls during test.

Tracker The size of the tracks data structure Figure 13 on page 61 and the final size of the internal stack.

Parser The average size of all AST structures of all the modules parsed.

Total The sum of the tracker and parser sizes.

No.	Stack	Tracked Paths				AST			Timings	Size		
	Max. Depth	Shortest [line]	Average [line]	Longest [line]	Number	Min [Node]	Avg [Node]	Max [Node]	Measure.	Tracker [Byte]	Parser [Byte]	Total [Byte]
0	11	1	39	86	44	3	471	2704	63	131177	797	131974
1	11	1	39	86	40	3	501	2704	61	128145	797	128942
2	9	1	39	86	36	3	636	5222	71	141972	1662107	1804079
3	11	1	39	88	56	3	674	5222	97	152717	2791506	2944223
4	10	1	39	88	46	3	625	5222	89	148889	2768766	2917655
5	8	1	39	86	28	3	399	1259	43	90738	259102	349840
6	6	1	40	86	14	3	366	933	12	16816	109	16925
7	10	1	40	88	58	3	680	5222	98	150576	3109952	3260528
8	6	1	39	86	12	3	414	933	17	25379	797	26176
9	11	1	40	88	44	3	508	1764	91	148898	2095278	2244176
10	18	1	42	139	164	3	390	4159	296	4051322	109	4051431
11	23	1	45	95	86	3	429	1705	169	893125	797	893922
12	11	1	43	92	30	8	410	1259	73	84676	797	85473
13	10	1	53	190	22	8	340	933	51	33963	797	34760
14	9	1	43	95	20	8	325	933	37	24365	797	25162

No.	Stack	Tracked Paths				AST			Timings	Size		
	Max. Depth	Shortest [line]	Average [line]	Longest [line]	Entries	Min [Node]	Avg [Node]	Max [Node]	Measure.	Tracker [Byte]	Parser [Byte]	Total [Byte]
15	11	1	87	820	28	8	537	2914	98	56473	797	57270
16	21	1	42	99	96	3	388	1705	150	1095409	1469	1096878
17	12	1	41	93	56	3	466	1705	111	216642	797	217439
18	7	1	41	93	10	3	366	719	10	17670	109	17779
19	1	1	54	107	2	83	83	83	1	5024	109	5133
20	12	1	53	120	40	8	322	933	121	123936	109	124045
21	14	1	42	116	86	3	277	1259	117	3330244	1453	3331697
22	4	1	42	102	10	8	357	933	6	7403	109	7512
23	1	1	49	97	2	79	79	79	1	5009	109	5118
24	1	1	49	98	2	74	74	74	1	5010	109	5119
25	1	1	50	100	2	272	272	272	1	5012	109	5121
26	1	1	51	102	2	597	597	597	1	5014	109	5123
27	1	1	51	101	2	82	82	82	1	5013	109	5122
28	1	1	46	92	2	703	703	703	1	5004	109	5113
29	1	1	54	107	2	1165	1165	1165	1	5017	109	5126

No.	Stack	Tracked Paths				AST			Timings	Size		
	Max. Depth	Shortest [line]	Average [line]	Longest [line]	Entries	Min [Node]	Avg [Node]	Max [Node]	Measure.	Tracker [Byte]	Parser [Byte]	Total [Byte]
30	1	1	52	103	2	307	307	307	1	5013	109	5122
31	1	1	49	98	2	1305	1305	1305	1	5008	109	5117
32	1	1	53	105	2	94	94	94	1	5015	109	5124
33	1	1	49	97	2	542	542	542	1	5007	109	5116
34	1	1	48	96	2	113	113	113	1	5006	109	5115
35	1	1	50	100	2	430	430	430	1	5010	109	5119
36	1	1	61	122	2	161	161	161	1	5032	109	5141
37	1	1	55	109	2	93	93	93	1	5019	109	5128
38	1	1	55	109	2	54	54	54	1	5019	109	5128
39	1	1	52	103	2	159	159	159	1	5013	109	5122
40	1	1	66	131	2	161	161	161	1	5041	109	5150
41	1	1	54	108	2	580	580	580	1	5018	109	5127
42	1	1	49	97	2	203	203	203	1	5009	109	5118
43	3	1	42	98	4	282	298	314	5	9714	109	9823
44	1	1	61	121	2	225	225	225	1	5033	109	5142

No.	Stack	Tracked Paths				AST			Timings	Size		
	Max. Depth	Shortest [line]	Average [line]	Longest [line]	Entries	Min [Node]	Avg [Node]	Max [Node]	Measure.	Tracker [Byte]	Parser [Byte]	Total [Byte]
45	1	1	52	104	2	55	55	55	1	5016	109	5125
46	1	1	51	101	2	53	53	53	1	5013	109	5122
47	1	1	50	99	2	53	53	53	1	5011	109	5120
48	1	1	49	97	2	78	78	78	1	5009	109	5118
49	1	1	50	99	2	122	122	122	1	5011	109	5120
50	1	1	50	100	2	702	702	702	1	5012	109	5121
51	1	1	61	121	2	86	86	86	1	5033	109	5142
52	3	1	41	93	4	111	196	282	5	9709	109	9818
53	1	1	51	102	2	252	252	252	1	5014	109	5123
54	3	1	47	116	4	86	184	282	5	9732	109	9841
55	1	1	46	91	2	147	147	147	1	5008	109	5117
56	1	1	44	88	2	209	209	209	1	5005	109	5114
57	1	1	47	93	2	587	587	587	1	5010	109	5119
58	1	1	59	118	2	194	194	194	1	5035	109	5144
59	1	1	55	110	2	167	167	167	1	5027	109	5136

No.	Stack	Tracked Paths				AST			Timings	Size		
	Max. Depth	Shortest [line]	Average [line]	Longest [line]	Entries	Min [Node]	Avg [Node]	Max [Node]	Measure.	Tracker [Byte]	Parser [Byte]	Total [Byte]
60	1	1	53	106	2	203	203	203	1	5023	109	5132
61	1	1	56	112	2	167	167	167	1	5029	109	5138
62	1	1	48	95	2	264	264	264	1	5012	109	5121
63	1	1	58	116	2	884	884	884	1	5033	109	5142
64	1	1	60	119	2	231	231	231	1	5036	109	5145
65	1	1	46	91	2	451	451	451	1	5008	109	5117
66	1	1	56	111	2	166	166	166	1	5028	109	5137
67	1	1	56	111	2	85	85	85	1	5028	109	5137
68	1	1	54	107	2	42	42	42	1	5024	109	5133
69	1	1	44	88	2	309	309	309	1	5005	109	5114
70	1	1	49	98	2	81	81	81	1	5015	109	5124
71	23	1	41	97	156	3	338	1705	223	1110612	5419585	6530197
72	9	1	42	99	32	3	331	1259	38	73611	797	74408
73	5	1	40	79	2	392	392	392	9	23516	797	24313
74	2	1	41	81	2	104	104	104	6	8182	2125	10307

No.	Stack	Tracked Paths				AST			Timings	Size		
	Max. Depth	Shortest [line]	Average [line]	Longest [line]	Entries	Min [Node]	Avg [Node]	Max [Node]	Measure.	Tracker [Byte]	Parser [Byte]	Total [Byte]
75	3	1	41	81	2	116	116	116	7	7327	781	8108
76	15	1	37	91	50	38	525	1259	78	1071710	1768145	2839855
77	15	1	37	94	50	38	518	1259	78	1009520	1692511	2702031
78	15	1	37	97	50	38	517	1259	78	1021900	1721937	2743837
79	15	1	37	90	50	38	519	1259	78	1002047	1659354	2661401
80	15	1	37	100	50	38	516	1259	78	1025111	1719269	2744380
81	15	1	37	89	50	38	527	1259	78	1031073	1631849	2662922
82	15	1	37	96	50	38	520	1259	78	1052999	1730721	2783720
83	15	1	37	90	50	38	517	1259	78	1002519	1755159	2757678
84	15	1	37	92	50	38	516	1259	78	1010450	1722151	2732601
85	15	1	37	90	48	38	574	1259	74	1031356	1769889	2801245
86	15	1	37	98	50	38	516	1259	78	1115946	1708960	2824906
87	15	1	37	88	48	38	539	1259	74	994072	1719794	2713866
88	15	1	37	91	50	38	517	1259	78	998839	1755142	2753981
89	15	1	37	86	50	38	521	1259	78	996238	1735659	2731897

No.	Stack	Tracked Paths				AST			Timings	Size		
	Max. Depth	Shortest [line]	Average [line]	Longest [line]	Entries	Min [Node]	Avg [Node]	Max [Node]	Measure.	Tracker [Byte]	Parser [Byte]	Total [Byte]
90	15	1	37	88	50	38	530	1259	78	999998	1782725	2782723
91	15	1	37	92	50	38	525	1259	78	1018057	1767643	2785700
92	15	1	37	94	50	38	526	1259	78	1001000	1768946	2769946
93	15	1	37	91	48	38	530	1259	74	999965	1691845	2691810
94	15	1	37	91	50	38	520	1259	78	996375	1758871	2755246
95	15	1	37	97	50	38	526	1259	78	1060385	1765758	2826143
96	15	1	37	97	50	38	520	1259	78	1020054	1732544	2752598
97	15	1	37	88	52	38	511	1259	79	1022419	1760221	2782640
98	15	1	37	92	50	38	527	1259	78	1016256	1770827	2787083
99	15	1	37	88	50	38	518	1259	78	999308	1755673	2754981
101	15	1	37	87	50	38	519	1259	78	1027062	1729776	2756838
102	15	1	37	91	50	38	517	1259	78	996310	1778043	2774353
103	15	1	37	90	50	38	519	1259	78	999318	1789287	2788605
104	15	1	37	93	50	38	524	1259	78	1007743	1765793	2773536
105	15	1	37	87	50	38	525	1259	78	999290	1762883	2762173

No.	Stack	Tracked Paths				AST			Timings	Size		
	Max. Depth	**Shortest [line]**	**Average [line]**	**Longest [line]**	**Entries**	**Min [Node]**	**Avg [Node]**	**Max [Node]**	**Measure.**	**Tracker [Byte]**	**Parser [Byte]**	**Total [Byte]**
106	15	1	37	92	50	38	522	1259	78	1015988	1762196	2778184
107	15	1	37	92	50	38	517	1259	78	998205	1684664	2682869
108	15	1	37	89	50	38	518	1259	78	998550	1723740	2722290
109	15	1	37	97	50	38	519	1259	78	1002789	1727730	2730519
110	15	1	37	91	50	38	524	1259	78	1005610	1801012	2806622
111	15	1	37	87	50	38	516	1259	78	1004049	1761308	2765357
112	15	1	37	95	50	38	516	1259	78	1015890	1719679	2735569
113	15	1	37	99	50	38	517	1259	78	1140723	1743644	2884367
114	15	1	37	87	50	38	518	1259	78	996013	1725473	2721486
115	15	1	37	97	50	38	517	1259	78	1001137	1720672	2721809
116	15	1	37	88	50	38	524	1259	78	1055317	1761675	2816992
117	15	1	37	86	50	38	523	1259	78	1007873	1754973	2762846
118	15	1	37	92	50	38	517	1259	78	1030991	1721096	2752087
119	15	1	37	86	48	38	578	1259	74	1086549	1862390	2948939
121	15	1	37	96	50	38	522	1259	78	1094117	1714981	2809098

No.	Stack	Tracked Paths				AST			Timings	Size		
	Max. Depth	Shortest [line]	Average [line]	Longest [line]	Entries	Min [Node]	Avg [Node]	Max [Node]	Measure.	Tracker [Byte]	Parser [Byte]	Total [Byte]
122	15	1	37	90	50	38	540	1259	78	1014350	1790506	2804856
123	15	1	37	86	50	38	521	1259	78	1004177	1563713	2567890
124	15	1	37	94	50	38	519	1259	78	997954	1729206	2727160
125	15	1	37	94	50	38	517	1259	78	1007385	1755158	2762543
126	15	1	37	97	50	38	518	1259	78	1033688	1725097	2758785
127	15	1	37	94	50	38	521	1259	78	1001089	1779440	2780529
128	15	1	37	93	50	38	521	1259	78	1014878	1761532	2776410
129	15	1	37	88	50	38	517	1259	78	988063	1722532	2710595
130	15	1	37	92	50	38	516	1259	78	999817	1720776	2720593
131	15	1	37	97	50	38	521	1259	78	999669	1740766	2740435
132	15	1	37	91	50	38	516	1259	78	999354	1718979	2718333
133	15	1	37	87	50	38	531	1259	78	988062	1773590	2761652
134	15	1	37	88	50	38	522	1259	78	996813	1716248	2713061
135	15	1	37	92	48	38	540	1259	74	1001741	1692876	2694617
136	15	1	37	88	50	38	522	1259	78	1002050	1681061	2683111

No.	Stack	Tracked Paths				AST			Timings	Size		
	Max. Depth	**Shortest [line]**	**Average [line]**	**Longest [line]**	**Entries**	**Min [Node]**	**Avg [Node]**	**Max [Node]**	**Measure.**	**Tracker [Byte]**	**Parser [Byte]**	**Total [Byte]**
137	15	1	37	89	50	38	519	1259	78	1051639	1757101	2808740
138	15	1	37	91	50	38	523	1259	78	1000116	1746135	2746251
139	15	1	37	87	50	38	528	1259	78	1009998	1758040	2768038
140	15	1	37	87	50	38	536	1259	78	1003115	1786431	2789546
141	15	1	37	93	48	38	582	1259	74	1049450	1871192	2920642
142	15	1	37	91	48	38	543	1259	74	1087473	1494554	2582027
143	15	1	37	93	50	38	516	1259	78	1100377	1730279	2830656
144	15	1	37	95	50	38	522	1259	78	1000660	1714085	2714745
145	10	1	40	93	32	3	412	1606	61	83247	2141	85388
146	10	1	40	86	34	3	336	1481	61	154710	109	154819
147	15	1	55	1345	90	3	665	8965	105	554496	109	554605
148	10	1	40	90	32	3	435	1481	57	163601	109	163710
149	10	1	43	97	28	3	442	1606	81	80573	2813	83386
150	11	1	41	88	34	3	507	1893	68	146961	2141	149102
151	1	1	50	99	2	349	349	349	1	5011	109	5120

No.	Stack	Tracked Paths				AST			Timings	Size		
	Max. Depth	Shortest [line]	Average [line]	Longest [line]	Entries	Min [Node]	Avg [Node]	Max [Node]	Measure.	Tracker [Byte]	Parser [Byte]	Total [Byte]
152	1	1	50	99	2	703	703	703	1	5011	109	5120
153	1	1	50	100	2	435	435	435	1	5012	109	5121
154	1	1	54	108	2	117	117	117	1	5020	109	5129
155	1	1	56	111	2	93	93	93	1	5023	109	5132
156	19	1	43	115	110	3	456	3815	186	1846250	109	1846359
157	18	1	44	115	110	3	455	3815	180	1836846	109	1836955
158	21	1	58	1241	110	3	461	3815	205	1838466	109	1838575
159	19	1	45	107	80	3	370	1315	106	5016174	109	5016283
160	11	1	40	88	40	38	555	1764	87	146324	1926364	2072688
161	11	1	50	138	28	8	315	1013	96	77178	797	77975

Table 7: Internals Evaluation

Table 6.2 on page 84 shows the sizes of the important data structures used by the new unit test framework. The most important information is the total memory size allocated for these data structures. It is up to 6.5 MB (start module No. 71). This result contradicts the bottleneck assumption made in Section 5.5 on page 57. What is also very interesting is the relation between the tracker memory size and the parser memory size. The tracker memory size depends on the amount of partial execution paths, the length of paths and the number of time measurements, which is equal to the number of function calls. The parser memory size shows the length of the source code, thus it is in direct relation to the code complexity. Comparing the tracker memory size column and the parser memory size column, we can see some degree of code reusability. This can be explained very simple. If the tracker memory size is large, then we can conclude that the whole execution path is long. On the other hand, if the parser memory size is relative small, then we know that the code is small. From these two facts we can conclude that the degree of the code reusability is the higher, the longer the exceution path and the smaller the code size is.

6.3 Time Measurement Precision

Duration The nominal execution time of `interval (time_in_ms)` function. The `interval` function is implemented in `code/Unit_Test_FW/tst/Times.py`. We did ten measurements for each nominal value.

Min. The minimal real execution time.

Avg. The average execution time.

Max. The maximum real execution time.

Abs. Diff. The absolute difference between the minimum and maximum execution times.

Rel. Diff. The relative difference between the minimum and maximum execution times relative to the average execution time.

Duration	Min. [ms]	Avg. [ms]	Max. [ms]	Abs. Diff. [ms]	Rel. Diff.[%]
1 ms	0,718	0,954	1,19	0,472	49
10 ms	9,595	9,879	10,163	0,568	6
100 ms	99,633	100,0415	100,45	0,817	0,8
1 s	999,571	999,889	1000,207	0,636	0,06
10 s	9999,655	10000,0045	10000,354	0,699	0,007

Table 8: ETM precision Evaluation

Table 8 shows the precision that can be achieved when measuring the execution time of a function. The most important column is the *Rel.*

Diff. column, which shows potential measurement error. However, the degree of precision that can be achieved in the new unit test framework is sufficient, because a precision degree expressed in seconds suffices to find performance bottlenecks.

6.4 Summary

This chapter covered all the metrics results calculated from the directory `projects/SW/lib/python`. These results were in the expected range. A lot of test cases failed due to the incomplete fixture maintained by the version control system.

The new unit test framework was also examined for performance bottlenecks. On the basis of the measurements carried out we can say that the assumption described in Section 5.5 on page 57 did not prove true.

The last point was an example that showed the precision of the execution time of a function.

7 Conclusion

The goal of this master's thesis was to design a new unit test framework with additional coverage metrics. For this purpose, a research on testing methods and coverage metrics were necessary. Before designing the new unit test framework we inspected the old one. The old framework used two testing methods - the unit test and the integration test. As far as the coverage metrics integrated in the framework concerns, there were two metrics - the line coverage (LC) and the code size - measured in SLOCs. Because the layout of the source code depended on the coding style of developers, the coverage results also depended on their coding style. In such a way the metrics results were not comparable.

The second weak point of this framework was the calculation of the metrics results. If a module was not executed directly from the framework but indirectly by another module, the executed lines of code in the module were ignored. As a consequence the modules executed in most cases indirectly, had much lower metrics results.

The next step was the research on testing methods. We found out that the usage of most of the testing methods depends only on the test case implementations. And therefore the quality of the testing methods are predetermined by the test case implementations and not by the testing methods themselves. The anomaly detection, the walk-through verification, and the mathematical verification could not be implemented as test case implementations. We also found out that the automated testing methods (the program spectra analysis and the adaptive test) were difficult to implement, since the exact input specification of software systems in *TTTech* (e.g. the user interfaces) were impossible to perform, due to a huge number of combinations of the input behavior.

To accomplish our task, we researched the code metrics which should measure the quality of the test case implementations. There were following three groups of code metrics: the control-flow code coverage metrics, the data-flow code coverage metrics, and the code complexity metrics.

Having collected the necessary knowledge, we designed the new unit test framework. This new framework should have implemented as many code metrics as possible. Unfortunately, this was not possible. From the control-flow code coverage metrics only the (SC), (DC) and the (LC) metrics were accepted to implementation, since the other metrics were too complex, concerning the company's requirements. Even the (SC) and the (DC) could not be implemented in the Python language due to the line resolution of the Python tracking system unless the coding guidelines existed. The data-flow code coverage metrics had to be completely refused, because the Python language is a dynamic language and therefore the necessary data for these metrics have been unavailable at compile time. The same problem occurred with the code complexity metrics as well. However, some code complexity metrics were chosen, because they could help to measure the static complexity of the source code. These metrics are the WMC, the Code Size, and the RFC. The last metrics, added to the new framework, was the ETM for measuring the execution times.

To solve the code coverage problem concerning the indirectly executed modules, the internal data structures were implemented in the new unit test framework. However, these internal data structures were expected to be a serious limitation in the situations when the new framework had to evaluate a huge number of Python modules.

The evaluation showed both the positive as well as the negative aspects of the new framework and of the test case implementations. The positive aspect was the fact that the size of the internal data structures was relatively small and therefore the new framework could evaluate a huge number of modules too.

As the evaluation of the test case implementations by the new unit test framework has shown, a noticable number of test case implementations which did not work, was discovered. Also the coding guidelines were broken at a number of locations. Paradoxically, these disclosed problems have been, in fact, the first fruit of the new unit test framework.

Glossary

AOSD Atomic Object State Diagram

AST Abstract Syntax Tree

CBO Coupling Between Object Classes

CC Condition Coverage metrics

COSD Composite Object State Diagram

DC Decision Coverage metrics

DCC Decision Condition Coverage metrics

DIT Depth of Inheritance Tree metrics

EDC Elementary Data Context

EDCM Elementary Data Context Metrics

ETM Execution Time per Method metrics

LC Line Coverage metrics

LCOM Lack of Cohesion in Method metrics

LCSAJ Linear Code Sequence And jump

MCC Multiple Condition Coverage metrics

MCDC Modified Condition Decision Coverage metrics

NOC Number Of Children metrics

ODC Ordered Data Context

ODCM Ordered Data Context Metrics

PC Path Coverage metrics

RFC Response For a Class metrics

SC Statement Coverage metrics

SLOC Single Line Of Code

TER Test Effectiveness Ratios

WMC Weighted Method per Class

References

[AB81] Dorothy M. Andrews and Jeoffrey P. Benson. An automated program testing methodology and its implementation. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 254–261, Piscataway, NJ, USA, 1981. IEEE Press.

[BBB$^+$04] Ralf Buschermöhle, Mark Brörkens, Ingo Brückner, Werner Damm, Wilhelm Hasselbring, Bernhard Josko, Christoph Schulte, and Thomas Wolf. Model checking grundlagen und praxiserfahrungen. *Informatik Spektrum*, April 2004.

[BI87] Farokh B. Bastani and S. Sitharama Iyengar. The effect of data structures on the logical complexity of programs. *Commun. ACM*, 30(3):250–259, 1987.

[BJK$^+$05] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer, July 2005. ISBN 3-540-26278-4.

[BN04] Bart Broekman and Edwin Notenboom. *Testing Embedded Software*. Addison-Wesley, 2004. ISBN 0-321-15986-1.

[CK94] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, Vol. 20(6):476–493, June 1994.

[CPRZ85] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A comparison of data flow path selection criteria. In *ICSE '85: Proceedings of the 8th international conference on Software engineering*, pages 244–251, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[Fag76] Michael E. Fagan. Design and code inspection to reduce errors in program development. *IBM Systems Journal*, pages 15(3):182–211, March 1976.

[FW88] Phyllis G. Frankl and Elaine J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, Vol. 14(10):1483–1498, October 1988.

[GS04] Christophe Gaston and Dirk Seifert. Evaluating coverage based testing. In Broy et al. [BJK$^+$05], pages 293–322. ISBN 3-540-26278-4.

[Gut99] Walter J. Gutjahr. Partition testing vs. random testing: The influence of uncertainty. *IEEE Trans. Softw. Eng.*, 25(5):661–674, 1999.

[Har00] Mary Jean Harrold. Testing: Roadmap. harrold@gatech.edu, June 2000. 22nd International Conference on Software Engineering.

[HRWY98] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical investigation of program spectra. In *PASTE '98: Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 83–90, New York, NY, USA, 1998. ACM Press.

[KSGH94] D.C. Kung, N. Suchak, J. Gao, and P. Hsia. On object state testing. In *COMPSAC'94 Conference*, Arlington, TX, USA, 1994.

[KST$^+$86] Joseph P. Kearney, Robert L. Sedlmeyer, William B. Thompson, Michael A. Gray, and Michael A. Adler. Software complexity measurement. *Commun. ACM*, 29(11):1044–1050, 1986.

[Lap92] Laprie, Jean-Claude. *Dependability: Basic Concepts and Terminology*. Springer-Verlag Wien-New York, 1992.

[Las82] Janusz Laski. On data flow guided program testing. *SIGPLAN Not.*, 17(9):62–71, 1982.

[Lin05] Johannes Link. *Softwaretests mit JUnit*. dpunkt.verlag, 2005. ISBN 3-89864-325-5.

[NF00] Martin Neil Norman Fenton. Software metrics: Roadmap. In The Future of Software Engineering, editor, *Anthony Finkelstein*, number ISBN 1-58113-253-0, pages 357–370, norman@dcs.qmw.ac.uk, martin@dcs.qmw.ac.uk, 2000. 22nd International Conference on Software Engineering, ACM Press.

[Nta84] Simeon C. Ntafos. An evaluation of required element testing strategies. In *ICSE '84: Proceedings of the 7th international conference on Software engineering*, pages 250–256, Piscataway, NJ, USA, 1984. IEEE Press.

[Nta88] S. C. Ntafos. A comparison of some structural testing strategies. *IEEE Trans. Softw. Eng.*, 14(6):868–874, 1988.

[Rus91] John Rushby. Measures and techniques for software quality assurance. Technical report, The National Aeronautics and Space Administration, September 1991. contract NAS1 17067.

[SL05] Andres Spillner and Tilo Linz. *Basiswissen Softwaretest*. dpunkt.verlag, 2005. ISBN 3-89864-358-1.

[VB01] Sergiy A. Vilkomir and Jonathan P. Bowen. Formalization of software testing criteria using the Z notation. In *COMPSAC '01: Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development*, pages 351–356, Washington, DC, USA, 2001. IEEE Computer Society.

[Vig05] Uwe Vigenschow. *Objektorientiertes Testen und Testautomatisierung in der Praxis*. dpunkt.verlag, 2005. ISBN 3-89864-305-0.

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

Index

Symbols

$CASE_DIR 4
$ROOT_DIR 4

A

abstract model 15
abstract syntax tree 38
  AST, 38, 41–43, 45–47, 49, 52, 55, 56, 59, 60, 81
  CONST, 38
  MODULE, 38
  NAME, 38
  SIZE, 61
  STMT, 38
  SUB, 38
AOSD 23
apriori knowledge criterion 14
assertion evaluator 23

B

basic test data 22, 23
boundary testing 29
branch 43
  branch, 49, 50
  branch point, 43
  false branch, 44
  true branch, 44
branch point 43
branch spectra 20
branch testing 32

C

c-use 30–32
class 34
  CoverageResults, 5, 6
  DocTest, 10
  Example, 10
  ModuleTestCase, 8
  TestCase, 7–9, 11
  TestLoader, 8
  TestResult, 9
  TestSuite, 9
  Trace, 5, 6
code complexity metrics . . . 27, 34, 36, 41, 42, 95, 96
  CBO, 35, 41, 42
  Code Size, 96
  code size metrics, 42, 45
  coupling between object classes, 35
  depth of inheritance tree, 35
  DIT, 35, 41, 61
  lack of cohesion in methods, 36
  LCOM, 36, 42
  NOC, 35, 41, 61
  number of children, 35
  response for a class, 35, 46, 67
  RFC, 35, 36, 42, 46, 61, 80, 96
  weighted method per class, 67
  weighted methods per class, 34, 45
  WMC, 34, 35, 41, 42, 45, 46, 61, 80, 96
code coverage 4, 6
code coverage metrics 27, 34
code metrics 27
code review 16, 21
code size 11, 67, 95
coder 21
cohesion 34
compiler 38
compiler.ast 38
complete-path spectrum 20
composite object state diagram 23
computational use occurrence . 30
condition 28
control flow graph 27–29
control-flow code coverage metrics 40, 95
  boundary-interior metrics, 29, 41
  branch condition coverage metrics, 28
  branch coverage metrics, 28
  CC, 28, 41
  code size metrics, 36
  condition coverage metrics, 28
  DC, 28, 41–45, 61, 80, 95
  DCC, 28, 41
  decision condition coverage metrics, 28
  decision coverage metrics, 28, 30, 41, 80
  LC, 45, 61, 80, 95
  line coverage metrics, 45, 80
  MCC, 28, 41
  MCDC, 29, 41
  modified condition decision coverage metrics, 29

multiple condition coverage metrics, 28, 29
path coverage metrics, 29
PC, 29
SC, 28, 41–43, 61, 62, 80, 95
statement coverage, 67
statement coverage metrics, 28, 30, 41, 80
structured path coverage metrics, 29
TER, 30, 41
test effectiveness ratios, 30
COSD 23, 24
coupling 34, 35
coverage file 3–7
coverage tool 5

D
data flow graph 30, 31, 41
data-dependence spectra 20
data-flow code coverage metrics41, 95
2-dr interaction, 33
all-c-uses some-p-uses metrics, 32
all-defs metrics, 32
all-du-path, 32
all-p-uses metrics, 32
all-p-uses some-c-uses metrics, 32
all-paths, 29
all-uses metrics, 32
EDCM, 33
elementary data context metrics, 33
ODCM, 33
ordered data context metrics, 33
required pairs metrics, 33
decision 28, 43, 44, 49
decision coverage 43, 58, 67
decision testing 32
def 30–32, 34, 41
definition 2
definition context 33
definition occurrence 30, 31
definition occurrence liveness . 31
designer 21
docstring 9, 10
docstrings 10
docTest 10
doctest 3, 5, 7, 10
doctest.py 3, 5, 10
docTestFinder 10
docTestParser 10
docTestRunner 10

E
EDC 33
elementary data context 33
end time 59
ETM . . . 37, 42, 45, 58, 61, 80, 96
event
call event, 39
exception event, 39
global scope event, 39
line event, 38, 39
local scope event, 39
return event, 39
Example 10
execution time per method 37
execution times per method . . . 68
execution-trace spectrum 20

F
fixture 8, 94
flat state machines 23
function name 39

G
globaltrace_cb() 5
gross execution time 37
gross time 45, 50, 59

I
independent path 58
inheritance tree 34
input criterion 14
input options
–debug, -g, 47
–details, -d, 47
–ignore, -i, 46
–label, -l, 47
–output, -o, 47
–verbose, -b, 47
–version, -v, 47
instance 34
interface 14
interior testing 29

K
k-dr interaction 33
k-tuple 33

L
LCSAJ 30
left 38
line 38, 39
line coverage 11, 67, 95

line number 39
linear code sequence and jump 30
lineno 38
live 31, 33
loadTestFromTestCase() 8

M

method 34
moderator 21
modtc1..N 8
modtc2 8
modtc2() 8
modtc2.run() 8
module name 39
ModuleTestCase 8
ModuleTestCase1..N 8

N

nature criterion 16
net execution time 37
net time 45, 50, 59
nightly test 5

O

obj() 8
obj.run() 8
object 2, 34
ODC 33
ordered data context 33
output spectrum 20

P

p-use 30–32
path coverage metrics 41
path spectra 20
Path testing 32
predicate use occurrence 30
program spectra 20
program spectrum 20
Python interpreter 5, 6, 10

R

reach 30–32
required pair 33
right 38
run() 8, 9
runTest() 7, 8

S

segment testing 32
setUp() 8
similarity 36, 42
single line of code 36
SLOC 11, 36, 42, 95
source 10
start gross time 59
start net time 59
statement 42
statement coverage 42, 58
statement testing 32
structured path coverage metrics 41
subclass 34
subsume 28, 29, 32, 34
sys 38, 39

T

tcfn 8
tearDown() 8
test 18
    adaptive test, 22, 23, 95
    adaptive tester, 23
    docstring test, 3, 7, 9–11, 46, 47, 52, 55, 61
    executable assertion, 22
    executable assertions, 22
    functional test, 19
    integration test, 24
    object state testing, 23
    random test, 18
    regression test, 18, 19
    structural test, 19
    symbolic execution, 19, 20
test case 1, 4, 6, 9–11, 14–16, 18, 19, 23, 27, 34, 94
test case function 1, 3, 6–8
test case implementation . . 1, 3–7, 11, 62, 95
test case string pair . 1, 3, 6, 9, 10
test driver 23
test results file 23
test suite 32
test_1..N() 8, 9
test_2() 8
tester 21
testing method . 13, 15, 17–20, 27
    adaptive testing method, 22, 23
    anomaly detection, 21, 95
    atomic object state diagram, 23
    automated testing method, 16, 18–20, 22, 24, 95
    black-box testing method, 15, 18, 19, 23, 24
    code size, 16
    difference-related testing method, 20

dynamic testing method, 16–18, 22
executable assertion, 23
executable assertions method, 22
formal verification, 16–18, 21–24
full testing method, 14
full testing testing method, 14
fully automated testing method, 16
functional testing method, 15, 17, 18, 23
gray-box testing method, 15, 19, 24
implementation stage testing method, 13
interface testing method, 15
manual testing method, 16, 19, 22, 24
mathematical verification, 21
partition testing method, 14, 19, 21
path testing metrics, 29
program spectra analysis, 20, 95
random testing method, 14, 19
static testing method, 16, 23
structural testing method, 15, 17, 18, 23
walk-through verification, 21
white-box testing method, 14, 15, 19, 20, 22, 24

testRunner 9
time.clock() 45
traceback 10
tracked 56
tst2 (2) 39

U

unit 24
unit test . . . 3, 7, 9–11, 24, 46, 47, 52, 55, 61
unit test framework . . 3, 5, 11, 26, 37, 38, 40, 42, 45, 46, 52, 60–62, 80, 93–95
Errors, 80
global_xml_file, 47
localtrace_cb(), 6
Metrics.py, 55
nightly-test.sh, 5
no_utest.txt, 4
Output.py, 55
Parser.py, 55, 59
test suite, 27–29, 32–34, 36
trace.py, 4, 5
Tracker.py, 55, 58
U_Test.py, 5
unit.py, 4–6
Unit_Test_FW_Tool.py, 46, 55
utest.txt, 3, 4
utest_coverage.txt, 3, 4

unittest 3, 5–7, 9
unittest.py 3, 5, 7
use 30–32, 34, 41
use occurrence 30

W

want 10
warning 80
while 39