Department of Computer Science

Andreas Lövgren, Thomas Persson

An investigation of component testing techniques in Base development and Connectivity Packet Platform development

D-level thesis (15 ECTS)

Date/Term:	2005-01-11
Supervisor:	Andreas Kassler
Examiner:	Donald F. Ross
Serial Number:	2005:12

This report is submitted in partial fulfillment of the requirements for the Master's degree in Computer Science. All material in this report which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Thomas Persson

Andreas Lövgren

Approved, 2005-01-11

Opponent: Kerstin Andersson

Advisor: Andreas Kassler

Examiner: Donald F. Ross

Abstract

The goal with the essay is to describe a common way of working regarding component testing on Base and Connectivity Packet Platform (CPP), two development departments at TietoEnator. From this research TietoEnator expects a guideline document, which describes the proposed way of testing. The company's main reason for wanting this research is that, by using the same strategies for testing, the testing departments will be able to make use of each other's work and it will also be easier moving personnel between the departments.

Structure, testing strategies, different tools and advantages/disadvantages with the different testing departments are analysed. The report aims to take the best parts from both areas and looks at what can be improved on respective department, in order to improve testing and reach a common way of working to the extent possible.

.

Contents

List of Figures

List of tables

1 Introduction

1.1 Background

TietoEnator is one of the leading suppliers of IT services to telecommunication and media sector in Europe, with focus on mobile solutions. TietoEnator offers a wide range of services including R&D-services, consult services, system development and integration for telecom and media sector. There are, among others, two development departments on TietoEnator called Base (see chapter 3.1) and CPP (Connectivity Packet Platform) (see chapter 3.2). Like other software producing companies, testing is an essential part of their development to reach a higher quality assurance and limit the amount of errors. Having a good testing environment is essential for high-quality and structured testing. Such testing environments provide tools to simplify test creation, a self-contained environment and test automation. The environment should basically perform everything except writing the tests. Component Test is the first test phase and is written and executed by the software designers writing the code to be tested.

Historically, the Signalling Base Development department has used a proprietary tool called Module Test Tool (MTT) for this purpose. While possessing many of the desirable characteristics of a component test environment (ease of use, ability to automate the running of large test suites, etc.), MTT had some drawbacks that prompted the development of a new component test strategy based on the X-Unit framework (see chapter 3.3.1), a framework for running tests, some years ago. The main problem that needed to be solved was that the turn-around time for running a test, finding a fault, correcting it, and running the test again was too long.

The first step was to automatically convert all the MTT test cases to X-Unit test cases in the C language. The test framework was then extended with stubs. The stubs are substitutes for external units called by the unit being tested. The interfaces of the stubs, including their names, are the same as the real units' interfaces. A simple stub can be an empty function replacing the real function, where only the function header is the same.

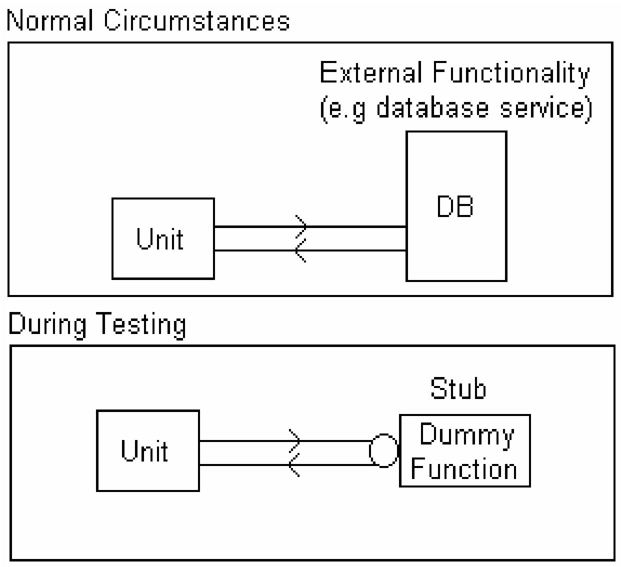


Figure 1.1: Stub replacing DB functionality

Stubs are often used for testing purposes when the external systems are not available.

The next step, which was taken for some of the more complex signaling modules, was to extract common functionality used by many test cases into a library called BB-lib, Black Box Library. Test cases written for execution in BB-lib are smaller and less complex than regular X-Unit test cases, since much of the module set-up and message verification has been moved into the library.

For CPP5, version 5 of the Connectivity Packet Platform, a decision was made to also move to X-Unit based testing. Prior to CPP5 all component testing was based on a platform called SimCello, which is basically comprised of a set of configurable simulators representing all the subsystems in CPP.

SimCello suffers from several drawbacks from a component-testing point of view. The SimCello platform is built into one large binary containing all the simulators and the test object. Just to build and link this binary takes a lot of time on anything but really high-end machines. This does not fit well with component testing where building and executing tests should be as quick as possible.

Tests are written in a script language, which stimulates a test driver written in C, which in turn operates on the test object. The script then looks in a log to see if the test object logged what was expected. This is less than optimal for several reasons; designers have to write tests in a language which is not the same as the test object is written in; there is a level of indirection between the tests (scripts) and the test object which is the C driver (which needs to be updated when the test object is updated). The tests check that the test object logs what is expected, not that it does what is expected.

There are dependencies both towards the simulators, which are developed by other subsystems, and towards the SimCello build support, which is quite complex. Interface or behavioural changes in subsystems have to be delivered before they may be tested by other subsystems.

White box testing is pretty much impossible. Without modifying the intermediary test driver, there is no way to test a specific section of code. The designer have to write test cases that send signals to the test object which, hopefully, will result in the intended section of code to execute. This makes it very hard to get coverage of some parts of the code.

The move to X-Unit based testing in CPP started off with creating simple stubs of the external interfaces used by the SS7-related modules developed in Karlstad. These stubs replace some of the simulators earlier found in SimCello. Then, the various teams started to build test cases for the SS7 components. In the beginning there were no direct guidelines available for this, which resulted in the component tests being somewhat dissimilar. Except for the stubs, which are shared by the component tests, there is no glue code or library (like BB-Lib in Signaling Base Development). CPP uses the Check X-Unit library directly.

1.2 Goals

The main goal is a common way of working regarding component testing for both Base development and CPP development. By using same system for testing, Base and CPP can make use of each other's work. It will also be easier moving personnel between the developments because they recognize the test code.

An investigation of the two testing environments is required to find out the differences between them and advantages/disadvantages with the strategies used on respective environment. This investigation will look at what can be improved and if it is possible to use the same way of working for both environments. In general, testing strategies and testing tools used should be analysed but focus should be aimed at the following areas.

Can the fundamental ideas and strengths of Black Box Library, the glue code library used on Base, be applied on CPP as well?

BTR-tool (Basic Test Report) is a tool used on Base to create test reports which show how well testing has been going. There are some problems with using this tool today, therefore BTR-tool should be investigated to try and solve the issues.

1.3 Working strategy

The research work is mainly based on interviews with employees on TietoEnator. Many employees at both departments have been interviewed and questioned regarding their testing environment. The cunning front people of each environment has been interviewed as well as common employees, in order to get everyone’s perspective on different matters. The main reason for the interviews was to gain knowledge of the testing environments in general and also their strengths and weaknesses. Most of the interviews were been recorded on tape in order to be able to extract most of the information brought up at the interviews. Aside from the interviews, parts of the testing code have been examined, in order to get a more detailed picture of the work of the testers. The work does not include any implementation efforts from the authors of the essay, this is left to the ones that should bring the ideas of this essay to reality.

2 Introduction to testing

In this chapter, the importance of testing, what testing means and different testing techniques are described.

2.1 Reasons for testing

Software products are mostly very complex, which increases the probability of bugs. One small error in a program can potentially make it worthless for the intended users. To minimize the chance of errors, testing is very important both during and after the development of a piece of software. If testing is not taken seriously, there is a great risk that program bugs make it to the market. This can result in time- and money consuming efforts and may also damage the reputation of a company. If a customer's business relies on a piece of software which fails then there could be repercussions such as lawsuits. Effective software testing helps to deliver quality software products that satisfy users' requirements, needs and expectations. If a piece of software is supposed to be run at an airport or aeroplane then it is extremely important that the program works according to its specification else it might endanger peoples lives.

Testing is both time and money consuming but in the long term it is the most economical choice. Very few organizations use a formal testing methodology. The majority treat testing as something that just happens after coding. Rather, if test planning begins early enough, and is treated as a parallel rather than a terminal effort, it can point out errors in interpreting specifications by the implementation team. What to test, how to test it, and who takes testing responsibility are all part of the testing methodology. [1,2.20]

2.2 Testing explained

There are two principal aims in testing software; defect detection and validation. The system probably has bugs in it and testing aims to find them, and subsequently correct them.

Testing can be used to demonstrate that the system fulfils its specification. Testing involves actions to find errors and faults in software and also actions to isolate errors during development. The main objective of testing is to help clearly describe system behaviour and to find defects in requirements, design, documentation and code as early as possible. Testing involves devising a set of inputs to a given piece of software that will cause the software to exercise some portion of its code. The developer of the software can then check that the results produced by the software are in accord with his or her expectations. Software testing is a process used to identify the correctness, completeness and quality of developed computer software. Actually, testing can never establish the correctness of computer software, as this can only be done by formal verification (and only when there is no mistake in the formal verification process). It can only find defects, not prove that there are none. In general, software engineers distinguish software faults and software failures. In case of a failure, the software does not do what the user expects. A fault is a programming error that may or may not actually manifest as a failure. A fault can also be described as an error in the correctness of the semantic of a computer program. A fault will become a failure if the exact computation conditions are met, one of them being that the faulty portion of computer software executes on the CPU. A fault can also turn into a failure when the software is ported to a different hardware platform or a different compiler, or when the software gets extended.

Software testing may be viewed as a sub-field of software quality assurance but typically exists independently (and there may be no SQA areas in some companies). In SQA, software process specialists and auditors take a broader view on software and its development. They examine and change the software engineering process itself to reduce the amount of faults that end up in the code or deliver faster.

Regardless of the methods used or level of formality involved the desired result of testing is a level of confidence in the software so that the developers are confident that the software has an acceptable defect rate. What constitutes an acceptable defect rate depends on the nature of the software. An arcade video game designed to simulate flying an airplane would presumably have a much higher tolerance for defects than software used to control an actual airliner.

A problem with software testing is that the number of defects in a software product can be very large, and the number of configurations of the product even larger. Bugs that occur infrequently are difficult to find in testing. A rule of thumb is that a system that is expected to function without faults for a certain length of time must have already been tested for at least that length of time. This has severe consequences for projects to write long-lived reliable software.

A common practice of software testing is that it is performed by an independent group of testers after finishing the software product and before it is shipped to the customer. This practice often results in the testing phase being used as project buffer to compensate for project delays. Another practice is to start software testing at the same moment the project starts and it is a continuous process until the project finishes. It is common knowledge that the earlier a defect is found, the cheaper it is to fix it. [3,4,12]

2.3 Test-Driven development

Test-driven development (TDD) is a programming technique heavily emphasized in Extreme Programming and is used by TietoEnator. Essentially the technique involves writing tests first then implementing the code to make them pass. The goal of TDD is to achieve rapid feedback. In order for TDD to work in practice, the system must be flexible enough to allow for automated testing of code. The system must also have testcases in place before TDD can be used. These tests must also be simple enough to return a simple true or false evaluation of correctness. These properties allow for rapid feedback of correctness and design.

It first begins with writing a test. In order to write a test, the specification and requirements must be clearly understood. The next step is to make the test pass by writing the code. This step forces the programmer to take the perspective of a client by seeing the code through its interfaces. This is the design driven part of TDD. The next step is to run the automated testcases and observe if they pass or fail. If they pass, the programmer can be guaranteed that the code meets the testcases written. If there are failures, the code did not meet the testcases.

The final step is the refactoring step and any code clean-up necessary will occur here. The testcases are then re-run and observed. The cycle will then repeat itself and start with either adding additional functionality or fixing any errors.

Despite the initial requirements, TDD can provide great value to building software better and faster. It offers more than just simple validation of correctness, but can also drive the design of a program. By focusing on the testcases first, the programmers must imagine how the functionality will be used by clients (in this case, the testcases). Therefore, the programmer is only concerned with the interface and not the implementation. This benefit is similar to Design by Contract but approaches it through testcases rather than mathematical assertions.

The power of TDD offers is the ability to take small steps when required. It allows a programmer to focus on the task at hand and often the first goal is to make the test pass. Exceptional cases and error handling are not considered initially. These extraneous circumstances are implemented after the main functionality has been achieved.

As a nice side-effect of TDD, the system will gain a large catalog of regression tests which can fully test the system at a moments notice.

A part of the Extreme Programming method is the unit testing concept. Various unit testing frameworks, based on a design by Kent Beck, have come to be known as xUnit testing frameworks and are available for many programming languages and development platforms. Unit testing is the building block to test driven development. Extreme Programming and most other methods use unit tests to perform black box testing. [1,12,14]

2.4 Testing levels

In this chapter, different levels of testing are presented. Testing occurs at every stage of system construction. The different levels of testing reflect that testing is not a single phase of the software lifecycle. It is a set of activities performed throughout the entire software lifecycle. This information is given to the reader to show that there are different levels of testing, but the focus on this essay is only Component Testing (the first level of testing). Facts are taken from [4] if nothing else is stated in the separate sub-chapters.

2.4.1 Unit/Component Testing

Unit (or Component) tests aim at testing a unit in isolation from the rest of the system. It may be a class or a cluster of classes that are tightly coupled, but it is always a conceptually atomic unit. For example, it might be the set of classes that provide the functionality of a binary tree. This would include an iterator class and a link class in addition to the tree class. The intent is that the coupling among internals of this component is more intense than the coupling of the component with any external objects. Unit testing is usually done by programmers. The main characteristic that distinguishes a unit is that it is small enough to test thoroughly, if not exhaustively. It is easier to locate and remove bugs at this level of testing. The objective of unit testing is to verify that individual units, the smallest compilable components, function correctly. Unit testing is also called component testing. However, component testing is sometimes considered to be a level of testing above unit testing. This may be the case with a system that contains individually testable components that are composed of multiple units. Unit tests are normally white box tests (see chapter 2.4.1.2).

It is important to realize that unit-testing will not catch every error in the program. By definition, it only tests the functionality of the units themselves. Therefore, it will not catch integration errors, performance problems and any other system-wide issues. Unit testing is only effective if it is used in conjunction with other software testing activities. [4,8,9].

2.4.2 Integration Testing

Integration testing exercises several units that have been combined to form a module, subsystem, or system. Integration testing focuses on the interfaces between units, to make sure the units work together. White box tests are used here because certain knowledge of the units is required to recognize if the combining of units was successful.

Integration tests can use stubs, which during testing replaces finished subroutines or subsystems. A stub might consist of a function header with no body, or it may read and return test data from a file, for instance.

2.4.3 External Function Testing

The external function test is a black box test (see 2.4.1.1) to verify that the system works according to the specifications. This phase is sometimes called an alpha test. Testers will run tests that they believe reflect the end use of the system.

2.4.4 System Testing

The system test is a more robust version of the *external function test*, and also called an alpha test. The main difference between system and external function testing is the test platform. In system testing, the platform is as close to production use in the customer's environment as possible. By putting the product in the customer's environment, it is easier to test, and be confident that the piece of software actually meets its requirements.

2.4.5 Acceptance Testing

An acceptance (or beta) test involves testing of a completed piece of software by a group of end users. The test is used to determine if the product is ready for deployment. Since the end users normally have a better idea of how the developed product will be used, the system will meet a more realistic testing phase than the system testers could offer.

2.4.6 Regression Testing

During development, there will be, at some stage, a working program that performs part of the final system functionality. As developing of the system proceeds, there is a risk of breaking the existing functionality. After any significant update to a program, existing test should be run again to see if the latest product changes have resulted in any new errors. This so-called regression testing is a good way to isolate errors.

2.5 Testing Methods

This chapter brings up some of the most common dynamic and static testing methods used. The dynamic methods discussed are Black Box testing and White Box testing and the static methods discussed are Code Review and Pair Programming. Facts are taken from [1], [4] and [20] if nothing else are stated in the sub-chapters.

2.5.1 Dynamic Testing

Dynamic testing is testing which involves software execution. In this sub-chapter, two common dynamic testing techniques, black box and white box testing, are discussed.

2.5.1.1 Black Box testing

Black box testing (also known as functional testing) is a testing technique whereby the tester does not examine the internal implementation of the program being executed. The method is used to validate that the software meets the requirements irrespective of the paths of execution taken to meet each requirement. Testing is performed against the specification and will discover faults of omission, indicating that part of the specification has not been fulfilled. The tester only knows the inputs and what the expected outcomes should be. The tester does not ever examine the programming code and does not need any further knowledge of the program other than its specification. The tester does not need to have knowledge of any specific programming languages to be able to write the test cases. The tester can also begin to design the test cases as soon as the specification is completed.

Black box testing is not 100 % reliable since it is unrealistic to test every possible input stream to the program it would take a inordinate amount of time; therefore, many program paths will go untested. Black box testing is not a technique where one writes a lot of test cases and hopes for the best. The aim of black box testing is to test as much as possible of the program, it cannot guarantee that all parts of the implementation have been tested.

Black box testing can be performed at the system level, external function level or unit level.

2.5.1.2 White Box testing

Also known as *glass box*, *structural*, *clear box* and *open box testing*. White box testing is performed to reveal problems within the structure of a program. This requires detailed knowledge of the internal structure of the program. A common goal of white-box testing is to ensure that every path of a program is exercised. A fundamental strength of white box testing is that the entire software implementation is taken into account during testing, which facilitates error detection even when the software specification is vague or incomplete. The effectiveness or thoroughness of white-box testing is commonly expressed in terms of test or code coverage metrics, which measure the fraction of code exercised by test cases. Common white box tests are *Statement testing, Loop testing, Path testing and Branch testing.* Statement testing involves tests of single statements. Loop testing test loops and the test includes skipping of loop completely, cause loop to be executed exactly once and cause loop to be executed more than once. Path testing makes sure that all paths in the program are executed. Branch testing makes sure that each possible outcome from a condition is tested at least once. White box tests are well suited in finding typo errors, logic errors and endless loops. White-box test cases are easily invalidated by changes in implementation details, but it results in very high levels of code coverage. White Box test can be applied on the unit and integration level.

2.5.1.3 Gray Box testing

Gray box testing means tests involving inputs and outputs, but test design is educated by information about the code or the program operation of a kind that would normally be out of scope of view of the tester. Test designed based on the knowledge of algorithm, internal states, architectures, or other high level descriptions of the program behaviour. Gray box testing examines the activity of back-end components during test case execution. There are two kinds of problems that can be encountered during gray box testing. One is when a component encounters a failure of some kind, causing the operation to be aborted. The user interface will typically indicate that an error has occurred. The other is when a test executes in full, but the content of the results is incorrect. Somewhere in the system, a component processed data incorrectly, causing the error in the results.

Even though testers normally do not have full knowledge of the internals of the product they are testing, a test strategy based partly on internals is a powerful idea. This is called gray box testing. The concept is simple: If you know something about how the product works on the inside, you can test it better from the outside. This is not to be confused with white box testing, which attempts to cover the internals of the product in detail. Gray box analysis combines white box techniques with black box input testing. In gray box mode, testing occurs from the outside of the product, just as with black box, but the testing choices are informed by the knowledge of how the underlying components operate and interact. Gray box testing is especially important with Web and Internet applications, because the Internet is built around loosely integrated components that connect via relatively well-defined interfaces. A good example of a simple gray box analysis is running a target program within a debugger and then supplying particular sets of inputs to the program. In this way, the program is exercised while the debugger is used to detect any failures or faulty behaviour. Rational's Purify (see chapter 3.3.2) is a commercial tool that can provide detailed runtime analysis focused on memory use and consumption. This is particularly important for C and C++ programs, in which memory problems are rampant.

2.5.2 Static Testing

Static testing is testing not including any software execution. In this sub chapter, two common static techniques, code review and pair programming, are discussed.

2.5.2.1 Code review

Code review involves that one or more team members read code for errors before it is added to the build. This requires developers having their reviewer/reviewers check the code according to a certain checklist. One criterion in the checklist can for example be “Are all variables initialised correctly?”. What criteria are chosen depends on what kind of business the list is being made for. For instance, criterions for testing of a game will not be equal to criteria for testing of airplane computer software. The list should only contain items that are frequent errors in that certain business. Not only the reviewer has the checklist but also the developer so that he/she knows what to keep in mind when writing the code.

2.5.2.2 Pair programming

Pair programming is a concept where two developers work at a single computer. One developer types the code while the other one looks for errors and improvements. The person writing the code is called *Driver* and the other is called *Partner*. The roles of *Driver* and *Partner* are switched frequently and also partners are switched throughout the project. By using this practice when programming, the code will be under constant review. Pair programming also distributes knowledge throughout the entire development team.

Table 2.1: On what levels Black Box, White Box and Code review can be applied.

Testinglevels	**Black Box**	**White Box**	**Code Review**
Unit Testing	+	+	+
Integration Testing	+	+	+
External Function Testing	+	-	+
System Testing	+	-	+
Regression Testing			
Acceptance Testing			

3 Overview of the departments

In this chapter, the signaling Base department and Connectivity Packet Platform (CPP) are presented. The chapter also discusses the relation between the two. Tools used for testing on the developments are described in the end of the chapter.

3.1 The signaling Base development

Base develops signaling systems, which is a collection of protocols for signaling in telecom networks. The protcols are defined by international standars ITU-T (Europe and Asia), China (China), ANSI (North America) and TTC (Japan). This collection of protocols can be viewed as a component, which can be used on different platforms for signaling. Signaling means all data that needs to be transferred in a telephony system in order to provide services to its users (traffic control, database connections, network management and basic call control). The signaling system, which Base develops needs to be as general as possible since it should be able to run on different platforms. Base signaling system is today used on CPP and Solaris, among other platforms.

3.2 Connectivity Packet Platform – CPP

The CPP (also called Cello) platform is built to support the multimedia services that characterize the third-generation cellular systems, such as WCDMA (Wideband Code-Division Multiple-Access) or UMTS (Universal Mobile Telecommunication System). The name Cello is derived from the ATM-cell, the cornerstone of ATM (Asynchronous Transfer Mode), a transfer protocol used in modern networking to combine data, voice and video on the same link.

The CPP platform can be developed into an ATM cell switching network node, for example a simple ATM switch, a Radio Base Station (RBS), IP router (Internet Protocol) or a Radio Network Controller (RNC).

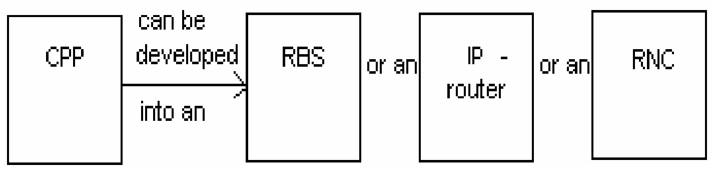


Figure 3.1 CPP - a generic platform

CPP provides tools and instructions to develop custom software and hardware for the nodes, such as radio algorithms, monitoring programs and device boards.

The CPP platform can be structurally divided into three main system areas within the target system, that is the core part, the Network Connection Handling (NCH) part and the Operation and Maintenance (O&M).

The core is responsible for maintaining software and hardware of a multiprocessor environment and provides methods for execution of application software and application hardware. O&M has the support function that an operator needs in order to control an application. NCH adds functionality to core and O&M, which makes CPP a network platform, based on ATM transport.

A CPP based node could for instance be an RBS, the node would then include the CPP platform and the application that runs on top of the platform.

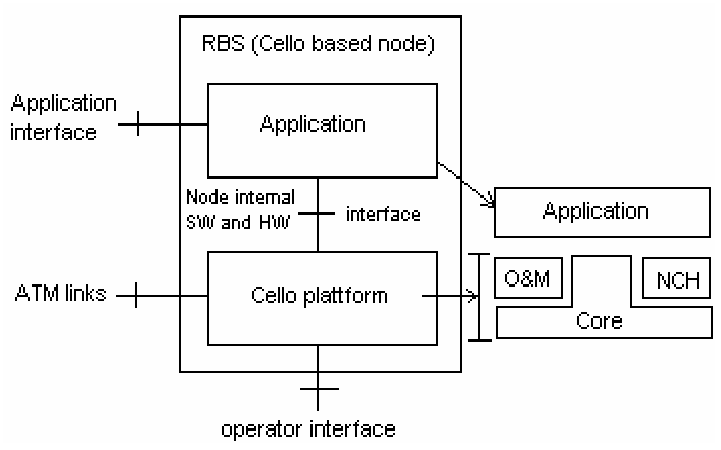


Figure 3.2: Cello node

The protocols developed on Base (see previous chapter 3.1) are used for sending signals. CPP builds functionality around these protocols in order to adapt them to their platform, so they can be used for signaling.

3.3 Testing tools

In this chapter, testing tools used on TietoEnator are presented. These tools are X-unit, Purify, PureCoverage, FlexeLint, Quantify and Doxygen.

3.3.1 X-Unit

X-unit is a simple framework and interactive tool used to structure and run unit test cases. There are a number of X-unit implementations; J-Unit is the X-unit implementation for testing of Java code, Check for C code and CppUnit for C++ code. The tool therefore makes it possible to write test cases in Java, C and C++. The tool is suited for companies developing code in Java, C or C++ since actual code and test code can then be written in the same language. In Check (which is the implementation used on Base and CPP), created tests are placed in test cases, which in turn are placed in suites. A complete component test build may contain several suites.

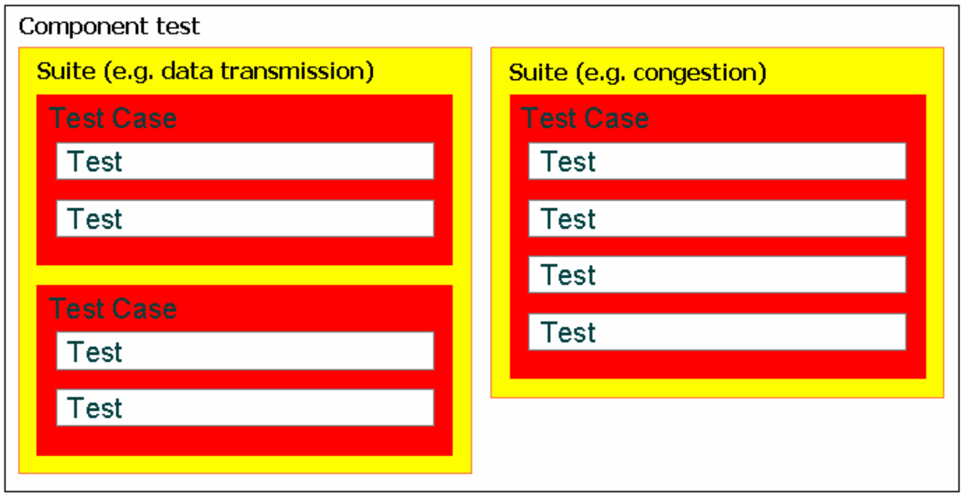


Figure 3.3 Component test structure in Check

The X-unit framework runs all test suites and also has the possibility to select individual suites to run, among the registered ones, so not all suites need to be run every time.

Check has built-in support for forking, which guarantees that every test case gets its own fresh execution environment. Test cases therefore will not affect each other in any way.

A concept, which often is used within testing is the red green bar, this comes from Junit, which can be operated through a graphical user interface. With Junit when the run button is pushed, the framework runs all suites, and if there are no errors the status bar remains green (If the bar is green the code is clean, said to the cadence of "When the light is green the trap is clean." from *GhostBusters, stated by [7]*). If an error is found the bar will turn red. Check and Cpp-unit are run through a command prompt. [6]

3.3.2 Purify

Purify is a run-time error detection tool for Windows and Unix platforms. The tool checks for memory leaks and other run-time errors by inserting object code before every load and store instruction. Purify can detect memory access errors, such as reading or writing beyond the bounds of an array, using un-initialised memory, reading or writing freed memory, reading or writing beyond the stack pointer, and reading or writing through null pointers. Furthermore potentially dangerous errors, such as calls to standard functions with incorrect parameters and potential core dumps can be detected.

Purify checks all the code in a 32-bit or 64-bit program, including any application, system, or third-party libraries. Purify supports the shared libraries `dlopen` (Solaris, IRIX, and 64-bit HP-UX) and `shl_load` (HP-UX), and works with complex software applications, including multi-threaded and multi-process programs. [5]

3.3.3 PureCoverage

PureCoverage measures program coverage by function, block or line. The tool has a customisable graphical user-interface, which is updated dynamically as tests are run. PureCoverage supports C, C++ and Java programming environments and can detect untested code with or without source code. PureCoverage provides code coverage information. This information can be used to assess how thoroughly testing has been done on C and C++ code, and to pinpoint the parts of the application that tests have not exercised. PureCoverage can also collect coverage data for Java applications running on a Solaris SPARC 32-bit Java virtual machine (JVM). PureCoverage monitors coverage in all the code in an application, even multi-threaded applications, including third-party libraries. Coverage data can be analysed in the interactive PureCoverage Viewer and Annotated Source window. Reports can be generated to help diagnose and handle testing deficiencies. [5]

3.3.4 FlexeLint

FlexeLint checks C/C++ source code and can detect bugs, glitches, inconsistencies, non-portable constructs and redundant code. The tool can check multiple modules. [5]

3.3.5 Quantify

Quantify is a performance profiling tool available for Windows NT and UNIX platforms.

It provides accurate and repeatable timing data. Quantify provides an analysis of applications run-time execution. Quantify identifies the portions of applications that dominate its execution time. Quantify gives information to eliminate performance problems so that software can run faster. It is possible to compare *before* and *after* runs to see the impact of the changes on performance. The 'Quantify'd Call Graph' graphically displays timing data, while the 'Function List' and 'Function Detail Windows' provide comprehensive timing data in tabular format. Each of these windows directs to areas in applications that are taking the most time to execute. [5]

3.3.6 Doxygen

Doxygen is a tool for generating documentation. With this tool, a CTS (Component Test Specification) can be generated automatically given a particular configuration specification for Clear Case (a version control system). The CTS will exactly match the build or delivery. Automatically generating the CTS will make sure that the CTS and the tested code are always in synchronization. The CTS can be generated as a hyper linked PDF or HTML. Commenting of test cases should be done carefully to get trace ability in trouble reports, requirements and other information. If the comments mention data types, functions, or any other element in the code, hyperlinks are automatically generated. When generating the HTML-version of the CTS, it’s possible to specify that the source code should be included in the result. This documentation makes a good reference for a component since the reader has access to both the documentation of the tests that describe how the component should work, and the actual test code that uses the component.

4 Analysis of Testing Environments

In this chapter, the testing environments used on Base development and CPP development are analysed. The aim of this chapter is to describe the different testing environments and to bring up strengths and weaknesses within each environment.

4.1 Base testing environment

In this sub-chapter, the testing environment used on Base development is discussed. The structure of the environment is presented, as are Black Box library and its use. Finally, tools related to the testing environment are described.

4.1.1 Structure

A complete testing structure on Base involves the module being tested, a *module-specific make file*, a *general make file* and all the related test cases together with Black Box library; a library, which supplies functionality to simplify creation of test cases. Every module-specific make file includes the general make file, which allows testers to run certain tools in combination with the test cases. These tools are PureCoverage, purify, FlexeLint and Quantify and they are used to get certain statistics over the testing performed, like code coverage.

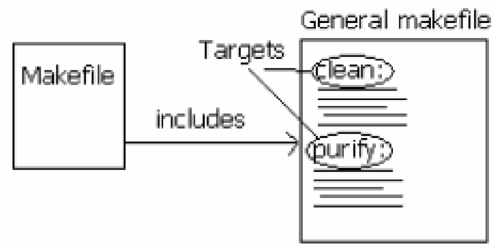


Figure 4.1 Make Targets

These tools are added as targets in the general make file, which makes it easy to compile and run test cases together with the desired tool. The company uses GNU-make (or Clear-make in GNU-compatible mode) and to build and execute the test cases the command ‘cmake test’ is used. To run one of the targets together with the test cases, the target is simply added to the command line, e.g. ‘cmake test purify’.

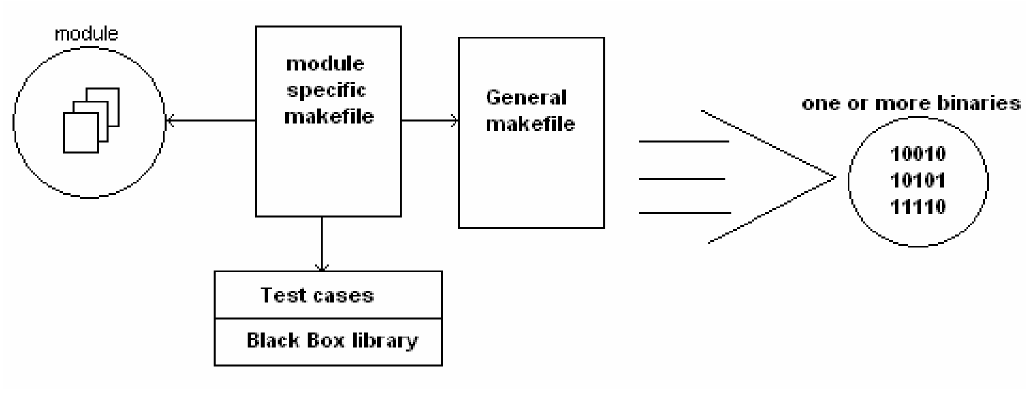


Figure 4.2: Build structure

The module-specific make file includes the module to be tested, the general make file, all associated test cases and also BB-lib. All these are linked together into one or more binaries. Therefore, the binary/binaries contains the module to be tested, the test cases and BB-lib.

X-unit (see chapter 3.3.1) is used to run the tests. Base uses Check, the C-version of X-unit, which allows forking. The results from tests are stored in a file called SS7tracelog.

4.1.2 Black Box Library

Black Box library (BB-lib) is the testing environment used on Base. BB-lib is a C++ library where common functionality used by test cases have been moved to. Test cases written in BB-lib are smaller and less complex than regular X-Unit test cases since much of the advanced instructions have been moved into help classes within the library. Having moved away advanced details and instructions has also raised the abstraction of test cases.

BB-lib supports functionality, which acts as stubs for the modules being tested. When messages are sent from the module being tested to some external functionality, such as ATM protocol, BB-lib answers with the message that the module expects to receive from its normal environment. This is possible since BB-lib contains stub functionality for all external interfaces used by the module being tested. Figure 4.3 illustrates a situation where the module (M3 in this case) sends a message destined to a certain service. BB-lib then responds with the message expected.

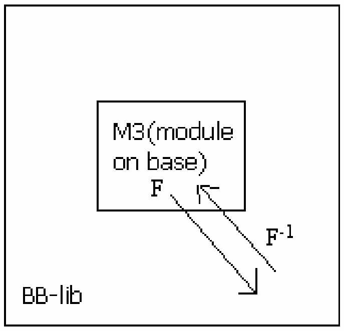


Figure 4.3: Base module sending a message to BB-lib which responds with the expected message.

The creation of configuration files, handling of different standards and handling of different versions are taken care of by BB-lib. These terms are discussed deeper in the following two sub-chapters.

4.1.3 Deficiencies prior to the introduction of Black Box library

Every test case requires a configuration and transport part, which initialises the correct state of the component being tested so that the test cases can test the behaviour it is supposed to. The configuration part configures a lot of default values for the tested component such as message timers and the transport part puts the component in the desired testing state, as an example a certain net topology.

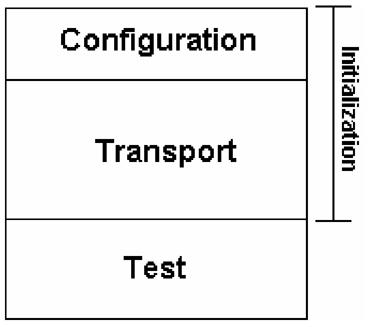
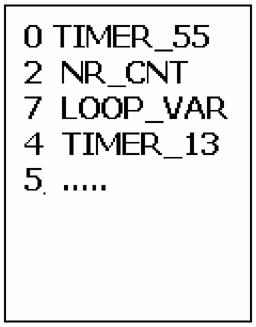


Figure 4.4: Initialisation of a test case.

Configuration and transport is done through configuration files on Base. Before the creation of BB-lib, testers had to set up theses configuration files manually. The work involved locating the necessary configuration files, which could be many, and perform the configurations needed to specify the correct behaviour and state of the tested component. This was a tedious and time-consuming process. The configuration files basically consisted, and still does, of a set of numbers. The numbers in the configuration files do not make sense to a human reader, which is why documentation labels were used so that testers could understand the meaning of all the different numbers.



```
0 TIMER_55
2 NR_CNT
7 LOOP_VAR
4 TIMER_13
5 .....
```

Figure 4.5: Configuration file

The problem with the labels was that there were no standard way to document them so the labels could often consist of less informative names, as illustrated in figure 4.5. This made changes in the files hard to perform.

Different configuration files had to be made for the existing standards ITU, ANSI, TTC and China because configuration had to be set up differently and messages did not look alike in the different standards.

There are different versions of a system and different versions often require different configurations. There might also be new features in new versions, which needs to be configured. This is why there was a need to supply different configuration files for different versions. Because of the complexity of the configuration files, much of the time that could have been used to create the actual test case was used to just get the module in the desired state.

The test cases were dependent on each other so test cases had other test cases as pre requirements. Testers often had to run several other test cases before the one they actually wanted to run, just to get the system in the desired configuration. The reason for the dependency is that a certain configuration setup was associated with each test case and test cases used the setups of the predecessor test cases and added needed modification. As a simplified example, setup of TC1 creates a link to test traffic over and TC2 is run after TC1 and adds another link. TC3 needs to test traffic over three links so TC1 and TC2 becomes pre requirement and setup of TC3 simply adds another link.

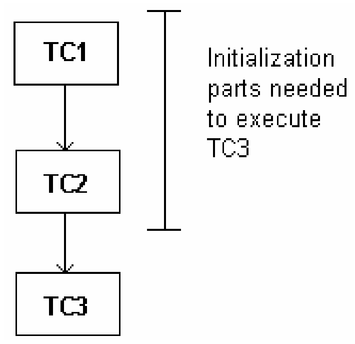


Figure 4.6: Test case dependencies.

So test cases always had to be run in a specific order. This made the testing process very inefficient and time-consuming. The number of test cases that had to be run before the target test case could often be many.

No time-control existed so test setups and test cases run successfully on one machine could fail on another because of performance differences between the machines. The reason for this is that many timers are involved in test runs, which are used when messages are sent between testing environment and tested module. A timer is started when a message is sent and the testing environment waits for a response from the tested module until the timer has expired. The timer is used because the system should not hang forever if the response would never arrive for some reason. If a timer was set on 100 ms for example when a message was sent from the testing environment to tested module and the response did not arrive until after 105 ms, the timer for the response would expire and an error would be reported. This is illustrated in figure 4.7 (TE in the picture stands for Testing Environment).

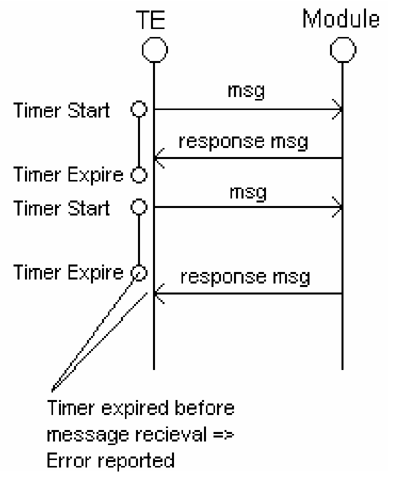


Figure 4.7: Timer expires before reception of message

The reason for the delayed response could be that the test was run on a slow computer or that the CPU was occupied with other actions at that time.

4.1.4 Testing strategies today

As mentioned in chapter 4.1.3, configuration files are used to initialize the tested module before the actual testing takes place. The initialization consists of two parts, configuration and transport. Chapter 4.1.3 also brings up some drawbacks with the configuration files.

The first step to improvement of the configuration files was to decide upon specific tags to be used in the files. This made it possible for an automated tool to take care of the configurations. As mentioned in the preceding chapter, this was hard to perform manually because the configuration files did not have any logical structure. However, the number of configuration files still were many, hence it was hard to keep track of them, even with this tool. Different files had to be created for the different standards and different versions. Today BB-lib takes care of the generation of configuration files. There still are a number of existing configuration files for different standards and versions, but these only contain default values, such as message timers. This means that the stored configuration files today only consist of the configuration part, not the transport part.

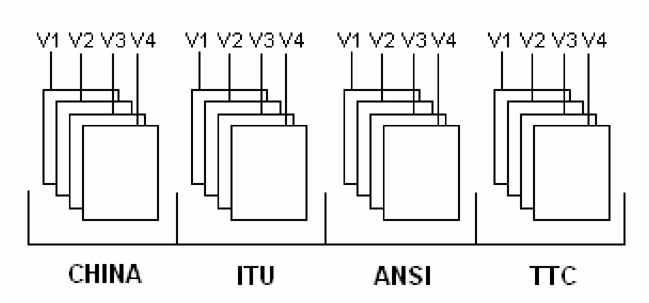
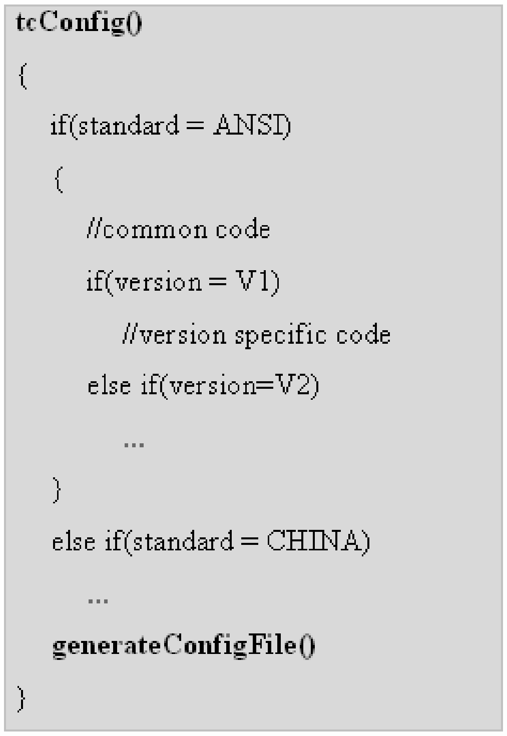


Figure 4.8: 4x4 default configuration files

The transport part, like net configuration, is specified in the test cases. BB-lib handles details involving the transport settings, therefore the setup code is short and lies on a high abstraction. A function call could for instance be addLink(), which adds a link in the net. Testers do not need to think about which standard is used or which version is used when creating test cases, only what to test and under which circumstances. Version and standard of the module to be tested is not decided until making the test build. BB-lib automatically configures the parts of the setup that are standard and version dependent. A simplified example of this is illustrated in the pseudo code below.



```
tcConfig()
{
   if(standard = ANSI)
   {
      //common code
      if(version = V1)
         //version specific code
      else if(version=V2)
         ...
   }
   else if(standard = CHINA)
      ...
   generateConfigFile()
}
```

Figure 4.9: Standard and version independent configuration file generation

In order to create the configuration file, which should be used by the module to be tested, BB-lib first chooses correct stored configuration file according to standard and version, for the configuration part.

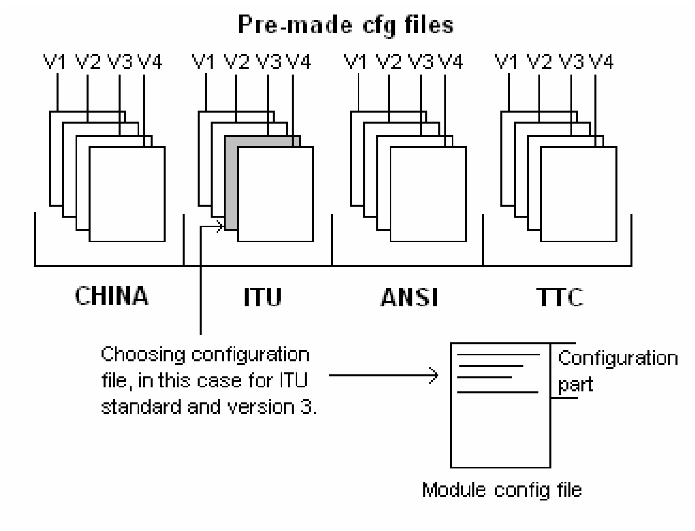


Figure 4.10: Choosing correct configuration file

The data in the configuration file is read and written to a new configuration file. The transport part (regarding net topology) is read from the test case and also written to the new configuration file. Also here BB-lib uses information about standard and version to configure details in the setup that is version and standard dependent.

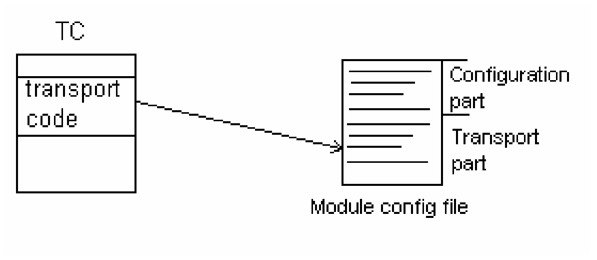


Figure 4.11: Transport code to module cfg file

When the two configuration parts have been written to the configuration file, it is given to the module to be tested.

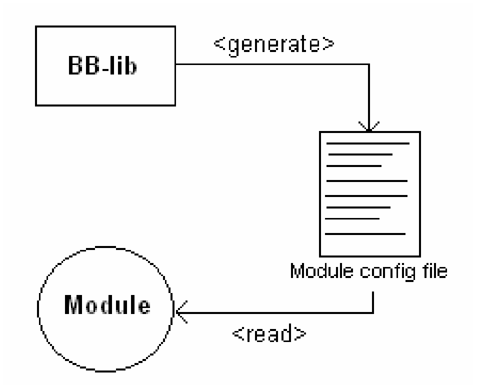


Figure 4.12: Module config file read by tested module

Since BB-lib has all configuration data given to the module to be tested stored in runtime variables, the state of the module is known to BB-lib. Since the state is known (net topology) BB-lib will always respond as expected on messages sent from the tested module, regardless of how the net is configured. Also, when writing a test case for a certain configuration, testers had to adapt the test cases according to the configuration. If for instance a certain net topology configuration consisted of two links, and the tester needed to start the traffic on the links, both links had to explicitly be started in the test case. An example of this is shown in the pseudo code below.

```
startLink(link1)
startLink(link2)
```

Figure 4.13: starting links

If the topology consisted of four links then four links had to explicitly be started instead. Today testers do not have to think about topology configuration when writing test cases. Since BB-lib has information about the topology, testers just need to call startAllTraffic() which starts traffic over all links, regardless of the number of links. BB-lib knows about existing links and therefore which links to start.

Today test cases are stand-alone. Test cases no longer use setup instructions from other test cases. As mentioned earlier in this chapter, BB-lib contains complete setup functions for all test cases that exist today. The result of this is that test cases no longer need to rely on other test cases being executed before them in order to reach a desired initialisation.

Testers now write their test cases in the C/C++ programming languages instead of scripts. This means that it is easier to make the test cases perform desired actions. Also, with the introduction of X-Unit (see chapter 3.3.1), testers can write both test cases and actual code in the same language. Before X-Unit, testers wrote test cases in a script language and the actual code in C/C++ or another programming language. It was therefore with the introduction of X-Unit that the creation of BB-lib was made possible. The script languages used before X-unit was too primitive to support development of a testing environment like BB-lib.

Before, regarding the usage of message timers, a function within the operating system was called by the tested module to start timers. The timer would then expire when the amount of time the timer was set to have passed. Today this OS function has been stubbed, so instead of allowing the module to call the OS function, it now calls the stub function. The only thing the stub function does is to put the timer variable, its value and the time when the timer was started in a table, as illustrated in figure 4.12.

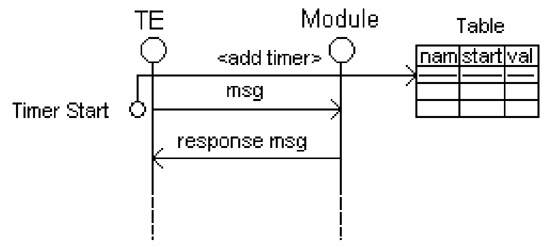


Figure 4.14: Timer put in table

The module automatically calls the timer start function, as it was before. The difference now is that the tester can decide exactly when a timer should expire. Since the OS timer never gets started, a timer will not ever expire until explicitly stated by the tester. This fact makes sure that performance differences between different machines do not affect the outcome of tests.

When the tester decides that the timer should expire, it is removed from the table, as illustrated in figure 4.13.

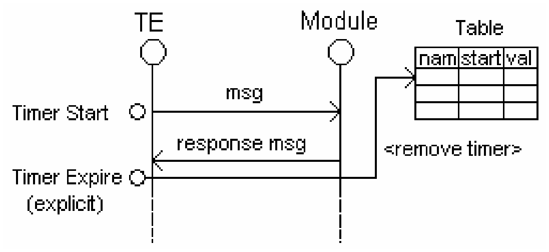


Figure 4.15: Timer removed from table

The table stores the timers in the same order as they should be expired. The usage of the table is that it is possible to make sure that testers do not expire timers in the wrong order. The expire function always picks the timer in turn. Still it is possible to choose to expire specific timers explicitly if desired.

Doxygen (see chapter 3.3.6) is available on Base and Doxygen comments are added in the test code. HTML and PDF documents can thereby be generated but this is not done today.

Often when testing is performed there is no systematic way in covering the tested product's functionality. Test cases are often measured in numbers, not what functionality they cover. The bag of test cases just keeps growing, but as a tester it is hard to tell what has actually been tested, if the question comes up.

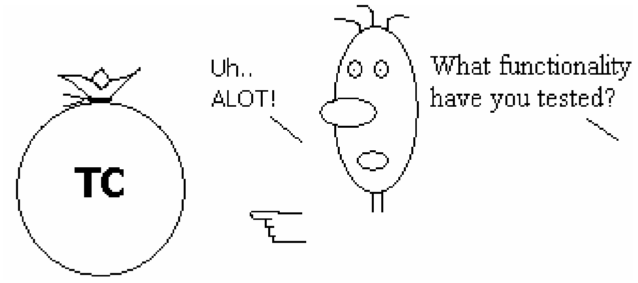


Figure 4.16: Functionality covering issues

In difference to this, Base has implemented a method to systematically cover functionality with the test cases. A matrix is used to get an overview of what functionality have been covered by tests. The matrix consists of a main matrix with sub matrices. Test cases within a matrix covers related functionality. All *states* that the module can be in and all *event*s that can occur within each state are covered.

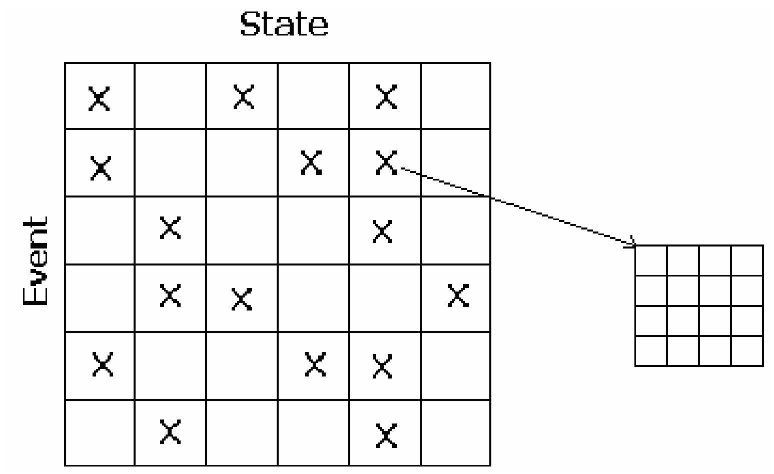


Figure 4.17: State-Event matrix

The goal is to test every state for all events. When a tester has written a new test case, the test needs to be registered to be able to show what functionality has been covered by that specific test case. When registering tests to X-unit, test cases related to a matrix are put in a separate suite. This is handled by the testing environment and nothing the testers need to take care of. Test cases therefore automatically get well structured in suites, which are registered to X-unit.

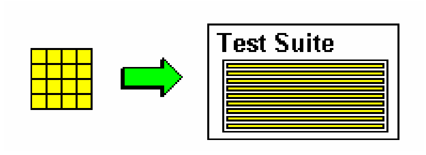


Figure 4.18: Matrix TC's to separate suite

Base is controlled by Way of Working, which is design rules and principles regarding testing. Within Way of Working the concept SQR (Software Quality Rank) is used to appreciate software/module quality. The ranking system goes from 0-3 and the goal for testers is to reach as high as possible. There are some rules associated with each rank to be able to reach it. As an example, a requirement can be that certain tools have been run against the code, such as PureCoverage to check that the level of code coverage is satisfying.

4.1.5 Associated Tools

This sub-chapter describes a number of tools that have been developed on Base since the development of Black Box library.

4.1.5.1 CNF-Builder

This tool is used to get access to Black Box library without having to write a test case. Through a CLI (Command Line Interface), commands can be executed. What the tool actually does is that it manipulates configuration files used for test cases. If, for instance, a setup is configured to have a certain amount of links, another link can be added directly from the command prompt or script.

4.1.5.2 Sesam

Sesam is a tool, which allows recreation of a test case execution by executing the file SS7trace.log, a log file generated from former test execution. This is possible because the log file contains information about all messages that were sent during the original test run. The log file works as a stub for the module involved since it knows what messages has been sent and therefore what messages the module expects to receive. Sesam has functionality for stepwise debugging, which means that errors can be traced. Sesam has an advantage over normal debuggers because it is possible to check exactly how the module was configured when the error occured, what state it was in, since it is an exact playback of what happened.

4.1.5.3 BTR-tool

BTR-tool is a shell-script developed on Base. It is used to get a report on test results received from PureCoverage, Purify, Quantify and FlexeLint (see chapter 3.3). These tools generate different files with the results from tests, e.g. amount of code coverage. The path to these files are specified in a make file so when this make file is run with the BTR-tool option, these files are automatically included in the BTR. The result is a BTR (Basic Test Report), including results from all the tools that have been run in combination with the test cases. The BTR is stored in XML format.

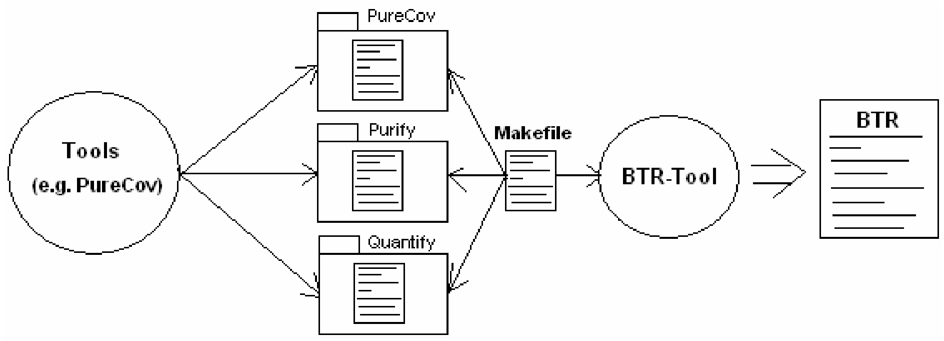


Figure 4.19: BTR (Basic Test Report)

The picture above, figure 4.17, illustrates result from PureCoverage, Purify and Quantify being included in a BTR generated by BTR-tool. The make file points at the paths to the results from the different tools, which BTR-tool uses.

In addition to generating the BTR, BTR-tool reports what features have been tested and what features have not. To every test completed, a tag, which identifies the covered test, is added. BTR-tool compares these tags with a list, where all required tests have been placed. An illustration of this can be viewed in figure 4.18.

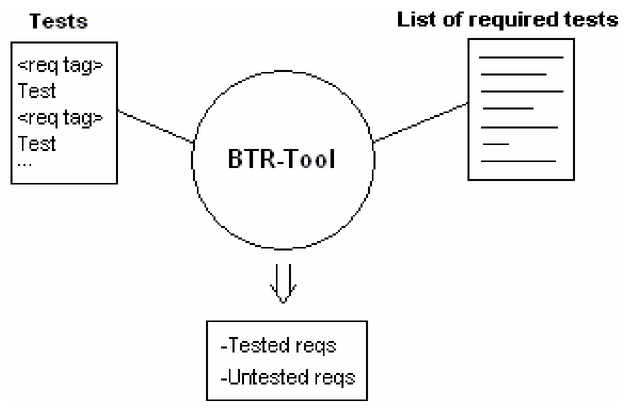


Figure 4.20: Tested/untested req. Report

It is important that testers do not forget to add the requirement tag as soon as a test has been written, otherwise BTR-tool will complain on this test being missing, even if it is not.

4.2 Connectivity Packet Platform testing environment

In this sub-chapter, the testing environment used on CPP development is discussed. The structure of the environment is presented as are the use of X-unit, suites and fixtures. The usage of Doxygen is also presented.

4.2.1 Structure

As on base, CPP's structure involves a component, a component-specific make file and a general make file. The make file for each component includes the general make file (see figure 4.1), which is used for all components. As on base, targets such as PureCoverage, purify, FlexeLint and quantify, among others exists in the general make file, which gives the option to run these tools in combination with tests. The component-specific make file specifies exactly what needs to be included, it points out specific C-files depending on which component it is and adds the specific functionality such as databases, nodes, etc.

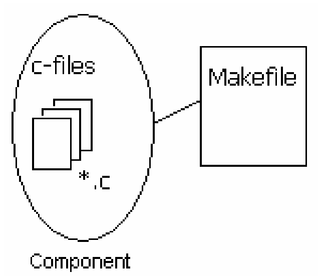


Figure 4.21: Component and its specific make file.

The make file for each of these components makes sure that the correct stubs are included to be able to satisfy the external dependencies of a component. A few generic stubs exist for common services like CLI (Command Line Interface) and DBI (Data Base Interface) that all component tests can use. The functionality in these stubs can be overridden by the tests if they wish to replace a particular piece of functionality to test some error condition for instance.

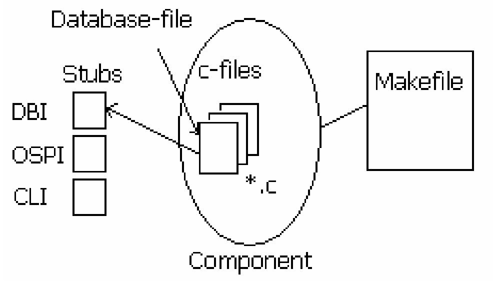


Figure 4.22: Stub functionality

Unlike base, where tests are run on Solaris, CPP tests are run on SoftOSE operating system, which in turn run on Solaris. SoftOSE, the test cases, the stubs and the component to be tested are all built into one or more binaries.

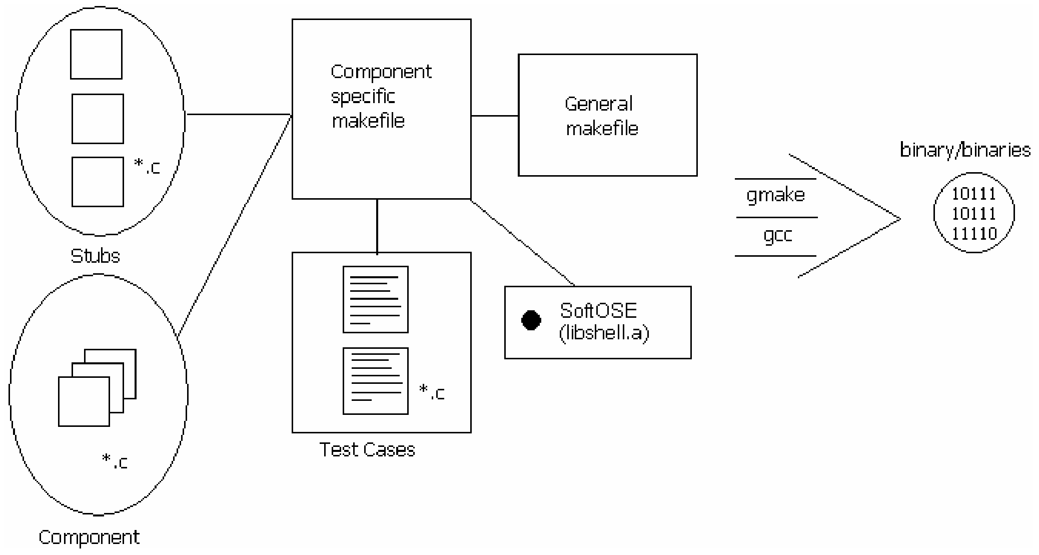


Figure 4.23: Make structure on CPP

The tests are built using make files and gmake, which builds exactly what is needed in order to perform the test. This helps keeping the build times and the test binary size down.

X-unit (see chapter 3.3.1) is used for running the tests. As on Base, the C-version of X-unit, Check, is used. When the test is finished reports from the tests are stored in a log file.

4.2.2 Testing strategies

Except for the stubs, which are shared by the component tests, there is no glue code or library (like BB-Lib in Signaling Base Development). CPP uses the Check (X-Unit) library directly. Test cases that are related to each other are registered in suites that may be executed separately once the SoftOSE shell, which is linked with the test binary, starts. It is therefore possible to run a subset of the tests. Testers themselves structure test cases in suites and register the suites to X-unit. This it different from Base, where the structuring of test cases into suites is handled automatically by the testing environment. On base, as mentioned in chapter 4.1.4, test cases related to one matrix are put into a separate suite. Since the various tests are concentrated to investigating a particular aspect of the test object, it is common that the test object needs to be in a particular state before the test is run. If each test had the responsibility of getting the test object into that state, the test code would quickly bloat and become more incomprehensible. On Base, BB-lib takes care of the complexity of dealing with configuration files, which are used by the test object to set up the desired testing state. CPP on the other hand does not use configuration files, so called *fixtures* take care of the configurations. A fixture consists of a setup function, which puts the test object in the correct state and a tear down function, which brings back the object to its initial state. A correct state varies depending on the test case, which results in a complete component test normally having several fixtures that may be combined to setup and tear down the test object for each individual test.

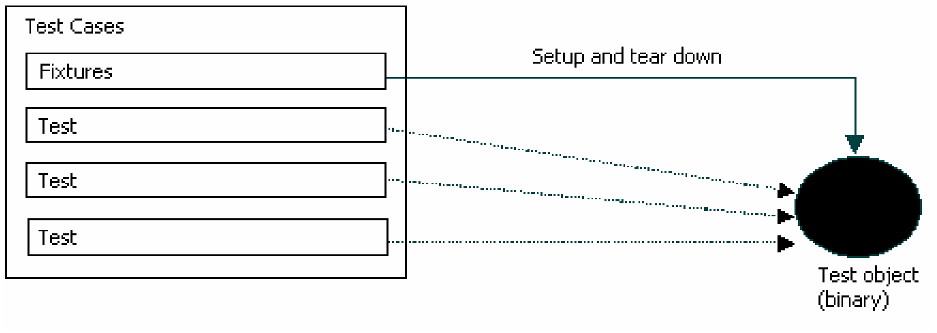
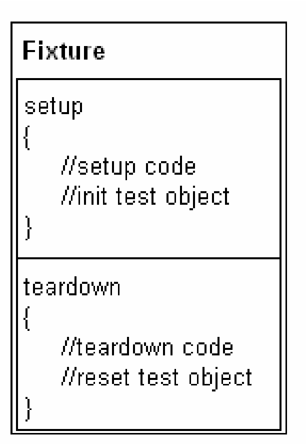


Figure 4.24: Test cases handled by fixtures

A fixture can for instance involve code to setup a number of communication links and a tear down of the same. The test itself might test that traffic runs as it should over the links and aim at testing traffic limits. The setup and tear down functions are ordinary C-functions as well as the tests.



```
Fixture
setup
{
    //setup code
    //init test object
}
teardown
{
    //teardown code
    //reset test object
}
```

Figure 4.25: Fixture

Fixtures is a general concept within X-Unit (see chapter 3.3.1) based testing and supported by many implementations, including Check, the C-version of X-unit used on CPP. Check associates one or more fixtures with a test case. The test case contains a couple of tests that all will have the setup and teardown functions of the fixture executed before respectively after the test is executed. The result of this is that tests themselves do not normally have to contain any logic to initialise the test object or remove any residue left in the test object after it has executed. Fixtures are normally shared between several tests, which means fixtures are good to reuse. Created fixtures are therefore stored for future use. Check allows a test case to contain several fixtures, they may be combined to create composite fixtures. Composite or combined fixtures can be used when there is a need for a certain fixture to initialise a test, but there already exist a fixture which contain most of the instructions needed for the new setup. The tester can then use the existing fixture and combine that fixture with a new fixture so that the precise setup needed can be achieved. As an example, if fixture 1 sets up five node communication links and test requires testing on a setup of nine links then a new fixture, which adds four links, can be created. These two fixtures can be used together to reach the desired state of nine communication links.

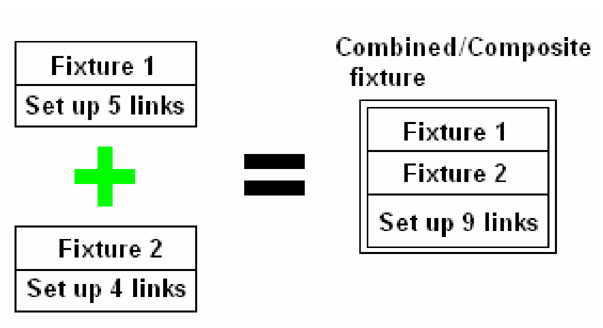


Figure 4.26: Combined/Composite fixture

Since CPP, as Base, uses X-unit it means that also testers on CPP has the possibility to write program code and test code in the same language, instead of scripts. Since tests only contain the logic related to the actual test and no setup code, tests are easier to write, read and understand. Tests could for example require that the test object contain the maximum allowed number of links to be available in the test object before they execute. The setup function for this particular fixture would then make sure that all the links are created and the tear down function would remove them.

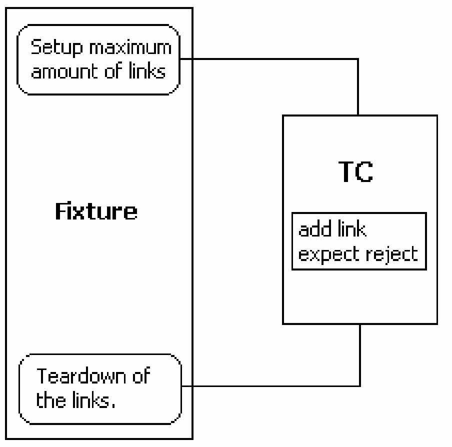


Figure 4.27: Test case and fixture

If the test is supposed to check what happens if the maximum number of links is exceeded, the only responsibility for the actual test case would be to simply add a link and expect a reject.

As on Base, test cases on CPP are stand-alone, no test cases need to be run before others to reach a desired configuration. This is because of the use of the fixtures, which has the full responsibility to set up the environments for the individual test cases.

Version dependent settings regarding the configuration files on Base is taken care of by BB-lib (see chapter 4.1.4). Testers therefore normally do not need to think about which version they are writing tests for. On CPP there are no automatic handling of version dependency. Fixtures and test cases need to be adapted manually according to version.

On base BB-lib performs configuration depending on standard. Testers on CPP do not normally need to handle different standards. This is mainly because CPP uses Base (see chapter 3.2), which takes care of the standard independent parts.

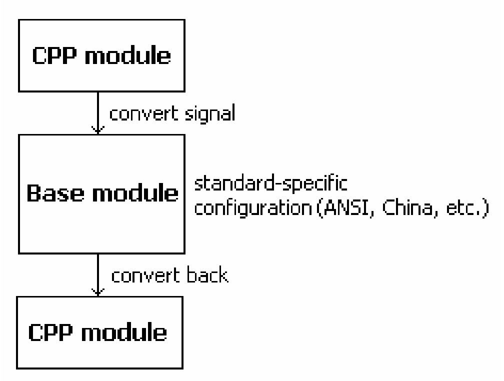


Figure 4.28: Standard dependent parts handled by Base.

Figure 4.26 shows an example where a CPP module sends a signal to a Base module. Before the signal is sent, it is converted to base signal structure so it can be understood by Base. The standard-specific configurations needed are taken care of by base. When the desired task is done, the signal is sent back to CPP, which receives the signal and converts it back. Some modules on CPP still needs to handle standard dependent parts however so testers can not completely overlook standard.

On CPP there is no fast way to check what functionality has been covered, it is hard for a tester to tell what has actually been tested. CPP does not have a matrix like base or anything likewise so there is no easy way to point out exactly what have been tested. CPP however has a good way of documenting the test cases. Since documentation is an essential part of TDD (Test Driven Development) and since one of the keys of TDD's work is to keep the time it takes from adding a test, running the test, adding the feature or fix, running the test again, and finally documenting the test, down to a minimum, the documentation also needs to be done as fast as possible. The old method was to maintain a Frame Maker document. A big drawback with the Frame Maker documents was that they needed to be kept synchronized with the actual test code. Today Doxygen (see chapter 3.3.6) is used. With this tool, a CTS (Component Test Specification) can be generated automatically from the test code. Automatically generating the CTS will make sure that the CTS and the tested code are always in synchronization and thereby eliminate the need to manually synchronize them. There is no easy way to find individual test documentations. If an error report comes from a customer, it is important to check if there is a test case that covers the specific error situation. On CPP testers normally have to look through a lot of documentation to find individual test documentation, which can be very time-consuming.

As mentioned in chapter 4.1.4, on Base, time no longer flows and testers themselves can decide when timers should start and expire. This way the results of tests do not differ on machines with different performance. On CPP, time still flows but since the test builds are very small, the outcome of tests does not vary on different machines. If a test build for some reason would get big enough, machine performance differences could affect the result of the tests.

CPP, as Base, is controlled by Way of Working (see chapter 4.1.4).

5 Improvement plan

In this chapter the criteria used for evaluation of testing strategies are explained as well as the proposed actions for improving testing. In the last part of the chapter the criteria are used to evaluate the actions proposed.

5.1 Evaluation criteria

In this sub chapter the criteria used for evaluation of testing strategies are explained.

5.1.1 Test creation

Test creation involves how much technical details are required and how much effort is needed to create tests. Writing a test should not require lots of work aside from writing the actual test case.

5.1.2 Test maintenance

This criterion looks at the effort required to maintain old tests when newer versions of systems are developed.

5.1.3 Test running

Test running involves the difficulty in choosing desired tests to run. The criterion also looks at whether or not tests are required to be run in a specific order to be executed as intended on the test object.

5.1.4 Functionality coverage

This criterion looks at how well tests cover the specified functionality of the system being tested. Having a good way of covering functionality with tests means that it is easier to prevent bugs and to assure that a certain feature is supported.

5.1.5 Automation

This criterion looks at how well testing is automated. Testers should not manually perform tasks that obviously can be handled by a tool.

5.1.6 Error detection

Fault detection/prevention involves how well testing techniques can find faults and/or prevent faults.

5.1.7 Common way of working

Common way of working is the criterion, which pervades the whole investigation. The criterion aims at having as equal testing methods as possible on Base development and CPP. As mentioned in chapter 1.2, having a common way of working will make it easier moving personnel between the testing departments and to make use of each other's work.

5.2 Possible improvements

This chapter analyses strategies and tools used for testing on Base and CPP. Both strengths and weaknesses are explained in order to see what ideas could be moved between the development areas and what needs to be improved. The chapter also aims at enabling use of same strategies on Base and CPP to the extent possible.

5.2.1 Testing strategies

In this subchapter, strengths and weaknesses of testing strategies used on Base and CPP are described and improvement suggestions are presented.

5.2.1.1 Configuration-files

Configuration on Base is, as mentioned in chapter 4.1.4, taken care of by Black Box library. Configuration files are created according to the setup instructions before the execution of a test case. The module to be tested reads the settings from configuration files to be able to setup the desired testing state. Configuration files are, however, still in use only because of backward compatibility with old test cases, which communicate the setup instructions to the module through files.

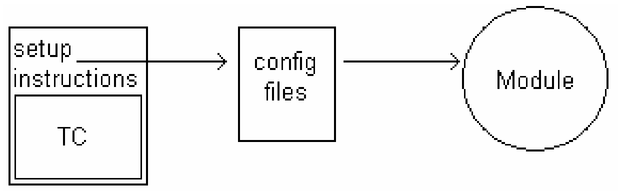


Figure 5.1: Configuration file generation

If all old test cases were to be converted, the generation of configuration files could be skipped completely and thereby all unnecessary intermediate steps would be avoided. Instead of delivering files to the module with setup instructions, test cases could instead give the instructions to the module directly. CPP, as mentioned in chapter 4.2.2, does not use configuration files at all. The testing environment performs configuration of the test object directly, which is handled by the fixtures used. All old test cases on CPP have been converted so there is no issue with backward compatibility. Doing same on Base will take some time since there are many test cases, but if the company works towards this, eventually all old test cases can be replaced by new ones and the usage of configuration files can be removed. Also, since the old tests are much more complex to understand than the new ones, maintenance of these tests are harder to perform. Therefore, converting old test cases will make maintenance of test cases easier. In addition to that, the old test cases has to be run in a separate framework and also might have to be run in an specific order, which is why it will also be easier to run the test cases.

5.2.1.2 Test setups/teardowns

BB-lib has been developed far in order to get a good structure and simplify testing on Base. BB-lib contains stub functionality and performs test case configuration regardless of standard and version. As mentioned in chapter 4.1.4, this is possible because BB-lib contains configuration settings for all versions and standards and automatically chooses correct configuration by checking current version/standard. On CPP test object does not need nearly as much configuration as on Base, so they also can lay most focus on the test setup part (e.g. net topology and traffic). On both Base and CPP, testers need to create setups, which are to be executed before the actual test in order to be able to test the desired state. On CPP the setups (together with the teardowns) are called fixtures, as mentioned in chapter 4.2.2. Fixtures are saved as they are created so they can be reused. When a tester on CPP writes a new test case, the first thing needed to be done is to check if a fixture already exists that fits the current test case. If no fitting fixture is found the tester needs to create a new one. There are no certain rules for naming saved fixtures, which is a problem. An approach to simplify the work on CPP is to use a well-structured hierarchy of fixtures, where fixtures have names decided according to some rule to reach uniformity and avoid overlapping fixtures. Fixtures should be catalogisized according to the version (for some parts of CPP also standard) it is designed for so they easily can be located.

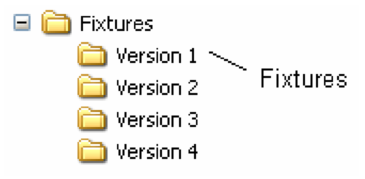


Figure 5.2: Structuring of fixtures.

The same thing could be done on Base, that testers begin to store their created setups for future reuse. To ensure a uniform non-overlapping amount of well-structured fixtures/setups, the best would be to assign one or more persons to perform the task. This would require alot of work from the persons assigned so from that perspective it would be best if testers could be ensured to follow the rules and guidelines for creating and storing fixtures/setups themselves. The problem with letting testers themselves create their fixture when needed is that there is a great chance that structure and names will not be uniform and thereby it will be hard to localize specific fixtures.

Another way to make the work easier for testers regarding setup is to make it more automatic. This can be done by putting related setups/fixtures in separate functions and within the functions use some conditional structure in order to choose correct setup/fixture depending on which version that is runing. The result of this is that testers would only need to specify the required setup function (and in the case of CPP also teardown function). For example calling a function createNodes would result in a call to the correct function for the present version. Pseudo code for this function is shown below.

```
function createNodes()
{
    if(V1)
        createNodesV1()
    else if(V2)
        createNodesV2()
    else if ....
}
```

Creation and structuring of setups/fixtures will require some time and effort but when it is done for a module it will be easier to write tests since locating and reusing setups/fixtures will be more straight forward. Working this way will also result in a more common way of working between Base and CPP.

5.2.1.3 Test case functionality coverage

To cover functionality, Base uses a state-event matrix, as mentioned in chapter 4.1.4. This is a good way to get an overview of covered and uncovered functionality. Since the matrix gives an overview of what functionality has not been tested, it results in more effective error prevention. The more functionality has been covered by tests the lesser is the chance of errors remaining unseen. CPP does not have any way to cover functionality like the matrix on Base. As mentioned in chapter 4.2.2, the only way to find out what functionality has been covered by test cases is to look at the documentation. There is no fast way to find out whether or not certain functionality has been covered by test cases. It is obvious that it would be very useful for CPP to also use a matrix for test case coverage overview. CPP however has a good way of documenting test cases using Doxygen, as mentioned in chapter 4.2.2. With Doxygen (see chapter 3.3.6) it is possible to generate a CTS (Component Test Specification) as a hyperlinked PDF or HTML. Together with the matrix the CTS's would become more useful since the matrix could work as a search engine for the documentation. If for instance an error occurs, the matrix is checked to see if there is a test case that covers that specific error. If there is a test case then it is easy to look it up in the documentation.

Base also uses Doxygen, as mentioned in chapter 4.1.4, but no CTS's are generated. Towards a common way of working and a more effective way of covering functionality CPP should work towards the usage of a matrix and on Base, Doxygen should be used to same extent used on CPP.

5.2.1.4 Structuring of test cases and suites

As mentioned in chapter 4.2.2, CPP uses the concept fixtures, where a fixture can work as setup for a number of test cases. The test cases are placed in suites and a test build can contain several suites. Testers themselves structure related test cases in separate suites. It is possible to execute individual suites, which means it is possible to just run some of the tests and not all of them each time.

On Base (see chapter 4.1.4), all test cases are registered in matrices, which in turn are registered in separate suites and then built together with the rest of the testing environment. In difference to CPP, testers on Base do not structure test cases into suites themselves, instead this is handled automatically by the testing environment. Every matrix becomes a separate suite, which is registered to X-unit. The advantage of having this automated is that the structure of the suites will have a logical and uniform structure (presuming that the test case matrices are well structured) and there will be no space for testers to perform their own structuring which probably will not always be uniform. It is possible to run sub sets of test cases, either a specific test case or a range of test cases.

Not every tester on CPP structures suites and test cases as they are supposed to with X-unit. It is not uncommon that there is a suite, which contains one single test case. Suites are meant to group test cases, which test related functionality, and if testers use suites this minimal way then much of the purpose of X-unit is lost. By implementing matrix to handle functionality coverage (as mentioned in chapter 4.1.4) CPP will automatically get a more uniform and better structure regarding what the suites should contain. If also the process of registering suites to X-unit is made automatic on CPP as it is on Base, a uniform and logical structure of test cases in suites is ensured. Computers do not have their own will so the instructions given will be performed exactly as expected (without any presence of bugs of course) but when the human hand is involved there is no way to be sure on how the result will look.

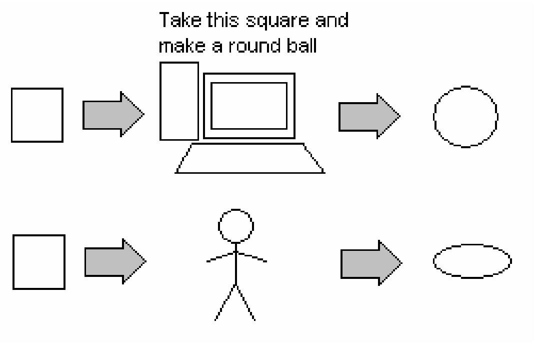


Figure 5.3: Do not trust humans.

One or more well skilled persons with good knowledge of the system should perform the grouping of test cases info matrices. The grouping should be done according to some predefined functionality specification. Doing so will make it easier for testers to check if certain functionality has been tested. As an example of this, on Base there is a function specification called SoC (State of Compliance), which tells exactly what should be supported. The SoC is split into chapters where each chapter covers related functionality. Using a guideline as the SoC when grouping test cases into the matrices results in a logical structure. Following the SoC/function specification makes it easier to run desired test cases. If there is a need to check if a module implementation supports the functionality described in a certain chapter in the SoC, the test case matrix covering that chapter can simply be run. If the test cases run through without errors, the functionality is supported. Since grouping according to a function specification will result in all described functionality being covered by tests, it also helps detect errors. Today, matrix test cases are not grouped according to the SoC on Base. The matrix test cases are grouped into functionality but not as logically structured as it would have been if SoC's had been used as guideline. Grouping test cases according to the SoC is an improvement that should be done on Base. There are function specifications on CPP corresponding to the SoC, which can be used when structuring matrix test cases.

5.2.1.5 Time control

The outcome of tests can vary on different computers because of differences in performance. As mentioned in chapter 4.1.3 this is because a lot of message timers are involved in testing. If a machine is fast enough or slow enough a timer can expire too soon or too late, which can make a test give an incorrect error report. Base solved this problem by implementing their system so that testers themselves can decide when timers should expire (see chapter 4.1.4). On CPP time still flows in the test cases, as mentioned in chapter 4.2.2. If CPP changes to the way Base work then testers will get better control of the test cases. Whether or not tests are run on a slow or fast machine will not affect the result of test cases, as on Base. It will also be easier to create tests since testers do not have to spend time making sure that timers don not expire before or after they should. Normally tests needs to be maintained often since computers performance increase fast which results in timers expiring at wrong times. With the implementation of time control, test cases will not need to be maintained for this reason anymore. Debugging will also be easier to perform when having control over timers.

Implementing time-control will be a big step towards a common way of working. A drawback with working this way is that testers will need to take care of timers in the test cases, which requires more effort. What is good with having to deal with the timers is that testers will get a more complete understanding of the module they are testing.

5.2.1.6 Simplicity of TC´s

There are some differences between Base and CPP regarding TC setup and teardown. Base uses fork when initializing a new test case (see chapter 4.1.1), which allows for each test case to execute in a new fresh environment. This way testers do not have to spend time coding teardown functions. Testers on CPP, on the other hand, need to write both setup and teardown functions. CPP runs upon SoftOSE operating system (as mentioned in chapter 4.2.1), which in turn run on Solaris. SoftOSE does not support forking of processes. SoftOSE is a real-time OS and this quality is needed by CPP's service so CPP cannot just run directly on Solaris to enable the usage of fork. Since it would save time to not be obligated to create teardown functions, an alternative could be to work towards usage of another real-time OS, which supports forking. Problem is that it is not trivial to switch from one OS to another, it would require a lot of effort and result in a lot of changes and there are not a vide range of real-time OS's. Because of the previous reasons, there is no trivial solution to reach a common way of working regarding this aspect.

5.2.2 Testing tools

This chapter discusses advantages and disadvantages with different tools. Whether or not these tools should be used on Base and CPP, is analysed.

5.2.2.1 CNF-builder

CNF-builder (see chapter 4.1.5.1) is a tool used on Base to manipulate configuration files without having to write a test case. There is no use for this tool on CPP since they no longer work with configuration files. If Base were to work towards removing the usage of configuration files there would be no use for CNF-builder either.

As mentioned before, to reach a more common way of working the configuration files should be removed from base.

5.2.2.2 Sesam

Sesam (see chapter 4.1.5.2) is a tool, which makes it possible to get a replay of a certain testing run by executing a log file. Sesam has an advantage over normal debuggers because it is possible to check exactly how the module was configured when the error occurred, what state it was in, since it is an exact playback of what happened.

BB-lib made the creation of Sesam possible since no time flow exists the time control in BB-lib has been stubbed out (see chapter 4.1.4). When no time flow exists it is possible to make an exact playback since timer values can be read from a file. Since no OS time function is called, the step vice debugging works; the timers will not expire after a certain amount of time.

Sesam is however almost unknown among the employees on TietoEnator and not in use at all even though it has been developed to work on Base. The developers of Sesam are certain that testers will have great advantage of using this tool. People on Base should lay some effort in trying to get Sesam in use. If Sesam proves to work well on Base then a corresponding tool could be developed for CPP as well.

Today a creation of a similar tool is not possible on CPP, but if the company decides to stub out the time on CPP as well, as suggested in chapter 5.2.1.5, an equivalent tool could be made. This would not only result in a more common way of working regarding the time control but would also result in both departments having an equivalent debugging tool.

5.2.2.3 BTR-Tool

BTR-tool (see chapter 4.1.5.3) is a tool used to create BTR's (Basic Test Report) and is written in a script language. The tool is used on Base to generate test results from PureCoverage, Purify, Quantify and FlexeLint. The BTR is stored in SGML format. BTR-tool has some drawbacks as it is today. Mainly it still has some bug issues to be solved, even though many of the bugs have been found and corrected. Another problem is that BTR-tool is not fully adapted for all modules it is used on. On the M3 module on Base, for instance, about thirty binaries are created from the test build. Each of these binaries will have their own separate reports for PureCoverage, Purify, Quantify and FlexeLint.

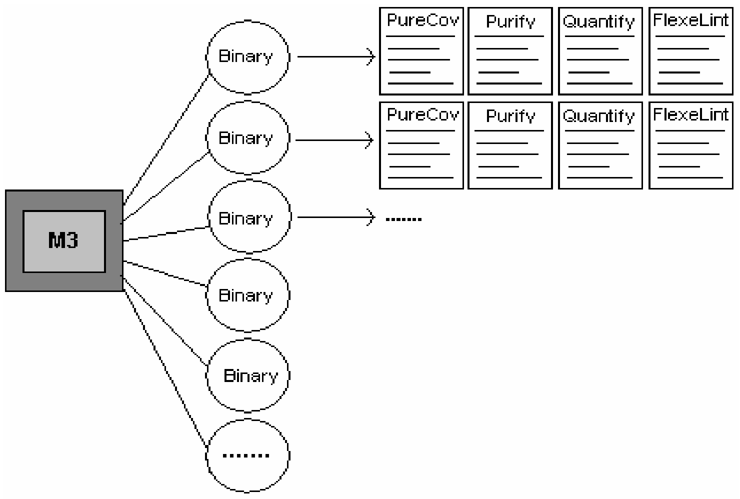


Figure 5.4: Report files for each binary on M3

BTR-tool only supports creating a BTR from reports for one binary at a time which means that BTR-tool must be run thirty times in order to get BTR's covering tests from all binaries.

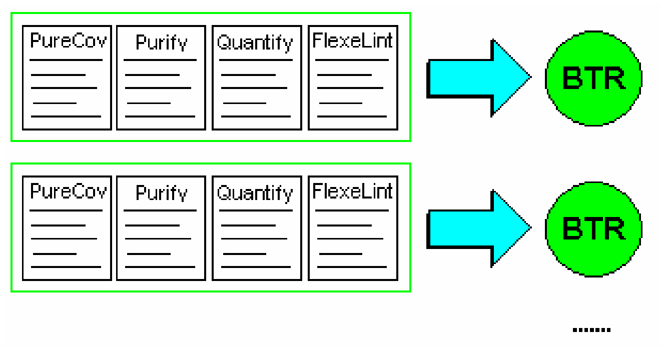


Figure 5.5: A BTR for every binary test report

All these BTR's are today manually merged into one document at the end.

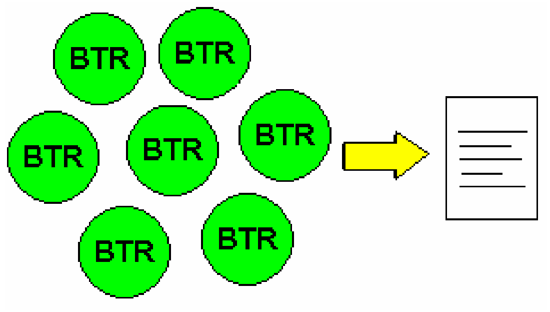


Figure 5.6: All BTR's merged into one document

A strategy to get rid of the need to run BTR-tool for each and every binary would be to merge all report documents of the same type (e.g. PureCoverage) into one document. Doing so would result in only having one document from each of the tools and not thirty.

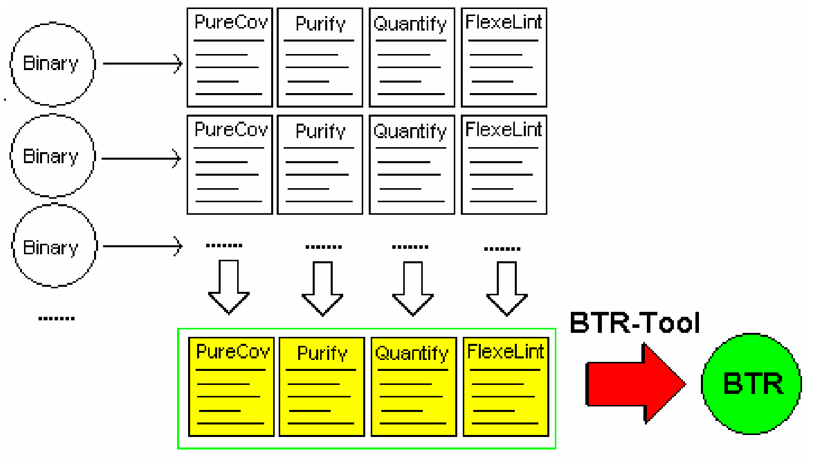


Figure 5.7: Merging report documents

This way BTR-tool will only be needed to run once to get a BTR covering all the binaries. The problem is merging these files, PureCoverage result files, for instance, are binary files specific for PureCoverage and these files cannot just be merged as text files can. There are however built-in functionality in PureCoverage, and the other tools, to merge the result files, but this has to be done manually after the tools are done with execution. This involves picking the result files and merging them, one by one. Because automization is important, an alternative is to create a script, which automatically performs the merging by running upon the tools and uses the built-in merge functions, as illustrated in figure 5.8.

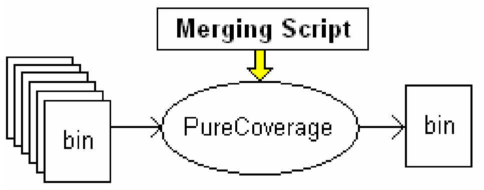


Figure 5.8: Merging script

The idea with BTR-tool is good, but as it is today, the tool only works for one binary. As it is today, reports from PureCoverage etc. are not created very often because it involves too much effort. If BTR-tool would function properly, BTR's would be created more often, which would help reaching a higher code quality. Changes need to be done before the tool can be of any essential use. On CPP there is no equal tool to BTR-tool, gathering of test reports is performed manually. As BTR-tool today does not function properly and because of that not much effort is saved using this tool, creating such a tool for CPP would not be worth the effort. If BTR-tool is changed so that it works well for Base, then developing a similar tool for CPP can be interesting, following guidelines from Base.

5.3 Advantages and costs of improvements

In chapter 5.2 some actions were suggested in order to improve testing on Base and CPP and to achieve a more common way of working between them. Following subchapters discusses the worth and cost of the suggested actions. Worth's and costs are estimated with a grade of 1-5. The last chapter summarizes the discussions and presents the result.

5.3.1 Explanation of the estimation values

The estimation of values in the following sub chapters is mainly performed by testers, which are involved in the different areas. Employees, who were working or had been working within Base, performed estimations regarding Base and the same way it was done for CPP related estimations. The testers performing the estimations were not randomly picked, they were well involved in their areas and all were front persons within respective department. This means that the reliability of these persons is high. Each of the employees gave a rating for the different actions, and if the ratings between the employees differed, a middle value was calculated. However, in many cases discussions led to a value that all could agree on, bringing up different views of the matter.

5.3.2 Convert old test cases (Base)

This action fulfils two criteria; test maintenance and test running. As mentioned in chapter 5.2.1.1, converting old test cases will make maintenance of test cases easier. This is because the old test cases are more complex to understand than the new ones. The assignment of maintaining old test cases of today is extremely time and effort demanding of the testers since there are many tests cases to maintain. Testers who have worked with maintaining old test cases rated the effort savings to four out of five regarding the test maintenance criterion. Since the old test cases has to be run in a separate framework and also might have to be run in an specific order, it will also be easier to run the test cases. All test cases can simply be run within same framework without having to think about which order the tests are run in. Testers who have worked with running old test cases and new test cases rates the effort savings to three out of five regarding the test running criterion.

There still are a lot of old test cases, converting all of these will therefore require quite a lot of time and effort. Involved testers estimated this cost to three out of five.

5.3.3 Structure setup/fixtures (general)

This action fulfils one criterion; test creation. Structure setup/fixture reuse will, as mentioned, ease test creation since the setup part can be reused. Since setups of different tests often are alike, testers will often be spared the work of creating setups and/or teardowns. However, setups/teardowns are only one part of test case creation, hence the rating do not get higher than it does. Testers who have worked with creating test cases rates the effort savings to three out of five regarding the test creation criterion.

The cost of this action is estimated fairly low since it mainly requires that testers follow certain rules when storing setups/fixtures, which will not require much more time or effort than of today. Existing fixtures/setups still needs to be structured however (if they should be able to be easy to reuse as well) so together with some testers the cost was estimated to two out of five.

5.3.4 Automate version and standard dependent setup/fixture selection (general)

This action fulfils three criteria; test creation, automation and common way of working. Since the action results in an easier way for testers to locate existing setups/fixtures, it results in easier test creation. Choosing the fixture/setup will be faster and also it will make testers more tempted to use this system with storing and reusing fixtures/setups. Until now many testers has not bothered with trying to reuse the fixtures because of the hardship in locating them. This action is a step further to the one discussed in 5.3.2, which is also aiming at addressing this problem. With this idea testers will not have to think about what version and standard they are dealing with, just what kind of setup/fixture they need. Testers who have worked with creating test cases rates the effort savings to three out of five regarding the test creation criterion. Automation is, as mentioned, viewed as an important aspect for effective testing. Testers who have worked with locating fixtures/setups for different versions and standards rated the effort savings to two out of five regarding the test automation criterion. Since the action proposed is general and directed at both Base and CPP, it results in a more common way of working. The setup/teardown part of test creation is not the largest part of a testers world so the testers who have worked with this rates the action to two out of five regarding the common way of working criterion.

The action will require someone to construct some kind of program, which automatically chooses correct fixture depending on standard and version. Except from creating the actual program, someone needs to be assigned to keep this software updated as new fixtures/setups are created. This person or persons will also have to make sure that the fixtures/setups are stored at correct locations and have specific names, if the program should work. This will require quite a lot of effort and together with some testers the cost of this action was estimated to four out of five.

5.3.5 Implement matrix (CPP)

The implement matrix action fulfils three criteria; functionality coverage, error detection and common way of working. Since the action results in a systematic way of covering functionality, it will make it easier to achieve good functionality coverage. Testers who have worked with functionality coverage rates the effort savings to five out of five regarding the functionality coverage criterion. Having a good way of covering functionality also results in better error detection since there is lesser risk for features to remain untested. Bad functionality coverage is a common reason for bugs making their way out of development. The involved testers rate the worth of this action to four out of five regarding the error detection criterion. If CPP would perform this change then it would result in a common way of working to a great extent since it is a big part of the testing. It would result a lot of changes.

The involved testers rated this action to five out of five regarding the common way of working criterion.

This action will require a lot of changes to the testing structure on CPP and the matrix strategy has to be implemented, which also is far from trivial. Involved testers rates the cost of this action to five out of five.

5.3.6 Register suites automatically to X-unit (CPP)

This action fulfils three criteria; test running, automation and common way of working. It simplifies the test running in the way that testers do not have to manually structure test cases into suites themselves. It will ensure that the structure will be uniform and logical, as long as the algorithm is well implemented. The involved tested rated the effort savings to one out of five regarding the test running criteria. The reason for the low rating is that registering suites is a small part of test running. Regarding the automation criterion the value of the action was rated to one out of five, this is because it is not that demanding task. Since Base already register suites automatically today, this action also fulfils the common way of working criterion. Also regarding this criterion, it will be a small-scale change hence the action is rated to one out of five by the involved testers.

To make this action a reality, one or more people need to be assigned to implement a program or routine, which handles the structuring and registration of test cases. Doing this will not be very complicated but will still require some time. A minor change of the current structure also needs to be performed. Involved testers rated this action to two out of five.

5.3.7 Group tests according to functionality specification (general)

This action fulfils two criteria; test running and error detection. The test running will be greatly improved since testers often want to run specific test to make sure certain functionality is supported/functions correctly. If the tests are grouped according to the functionality specification, the related tests can more easily be identified and run. Involved testers rates this action to four out of five regarding the test running criterion. The action affects error detection positively in the way that necessary tests can more easily chosen when grouping tests according to functionality specification. This results in lesser chance for bugs to remain unseen. This is only to a minor extent however since the bugs will often be detected when the entire test suite is run. Involved testers rates this action to one out of five regarding the error detection criterion.

The action will require one or more employees to regroup all or most of the test cases, which will require quite a lot of time and effort. The action will however only affect this area itself and the work will be fairly straightforward if the assigned people are careful. Involved testers rates the cost to three out of five.

5.3.8 Implement time control (CPP)

This action fulfils four criteria; test creation, test maintenance, error detection and common way of working. Test creation will be easier since the testers will not need to worry about whether or not timers will expire before they should (on other words, they need not worry about that an action may take longer than expected to execute). Involved testers rates this action to two out of five regarding the test creation criterion. The reason for the fairly low rating is that it is not that common that timers expire incorrectly on CPP. It is not that urgent problem as it was on Base, even though it still occurs. The action will improve error detection since it will enable the use of debuggers, since timers will no longer be involved. Above all, this enables the use of Sesam (see chapter 4.1.5.2). Involved testers rates this action to two out of five regarding the error detection criteria. A common way of working is achieved since this is the way it is working on Base. Involved testers rates this action to three out of five regarding the common way of working criterion. A fairly good rating because the action will result in both uniformity in the creation of tests and also uniformity in debugging (the level of common way of working this is however also depending on whether or not both departments are going to use Sesam).

This action will require changes in the way test cases work with timers, which will require some changes. These changes should however be straightforward and not very complex. The most demanding part of the work will be to change the existing test cases since testers will explicitly need to expire timers. The involved testers rate the cost of this action to three out of five.

5.3.9 Get Sesam in use (Base/general)

This action fulfils one criterion; error detection. This is because Sesam is a tool for step vice debugging which only needs a test log file as input. The tool makes it possible to view the exact module configuration when the error occurred. This possibility will greatly improve the ability to detect errors. Involved testers rate the worth of this action to three out of five.

The cost of getting Sesam in use is low, in essence, what is needed is to spread the tool among the employees and see to that they learn how to use the tool. What is mainly important is to explain the advantages with using the tool. The cost of this action is estimated to one oft of five.

5.3.10 Develop merging script for BTR-tool (Base)

This action fulfils one criterion; automation. Besides that the action also results in other qualities not measured by the criteria, mentioned in 5.2.2.3. However, since the work of merging all the result files from PureCoverage and the other tools would be very time and effort consuming, this action saves a lot of work. Involved testers rates this action to five out of five regarding the automation criterion.

Developing a merging script would be straightforward and would not require much time. A tester alone would be able to create such a program within days. The cost of this action is rated to two out of five.

5.3.11 Summary of the actions

In the table below the criteria fulfilled by the different actions are summarized. How well the criteria are fulfilled is estimated with a grade of 1-5, where 5 is the highest grade.

Table 5.1: Actions and the criteria they fulfil

Actions	**Criteria**						
	Test creation	**Test maintenance**	**Test running**	**Functionality coverage**	**Automation**	**Error detection**	**Common way of working**
Convert old test cases (Base)		**4**	**3**				
Structure setups/fixtures (general)	**3**						
Automate version and standard dependent setup/fixture selection (general)	**3**				**2**		**2**
Implement matrix (CPP)				**5**		**4**	**5**
Register suites automatically to X-unit (CPP)			**1**		**1**		**1**
Group tests according to functionality specification (general)			**4**			**1**	
Implement time-control (CPP)	**2**	**1**				**2**	**3**
Get Sesam in use (Base/general)						**3**	
Develop merging script for BTR-tool. (Base)					**5**		

Following table summarizes the cost estimation of the actions proposed. The estimation scale goes from 1 to 5, where 5 is the highest cost.

Table 5.2: Actions and their costs

Action	Cost
Convert old test cases	3
Structure fixtures /setups	2
Automate version and standard dependent fixture selection	4
Implement matrix	5
Register suites automatically to X-unit	2
Group tests according to functionality specification	3
Implement time-control	3
Get Sesam in use	1
Develop merging script for BTR-tool.	2

The last table relates the cost of an action with its relevance. This is calculated with the formula $\sum \mathbf{Ecv_n / Ec = E\eta}$ where $_n$ is the action.

Ecv = Estimated *criteria* value (see table 5.1)

Ec = Estimated *cost* (see table 5.2)

$E\eta$ = Relevance value of the *action*

The higher value $E\eta$ has implicates that the action is more valuable in relation to it´s cost.

Table 5.3: Calculation of relevance values

Action	Applying formula	Result ($E\eta$)
Convert old test cases	(4+3)/3	2.3
Structure fixtures/setups	3/2	1.5
Automate version and standard dependent fixture selection	(3+2+2)/4	1.8
Implement matrix	(5+4+5)/5	2.8
Register suites automatically to X-unit	(1+1+1)/2	1.5
Group tests according to functionality specification.	(4+1)/3	1.7
Implement time-control	(2+1+2+3)/3	2.7
Get Sesam in use	3/1	3
Develop merging script for BTR-tool.	5/2	2.5

According to the estimations and calculations, the actions are structured in the following table in the order of relevance. This means that the higher in the table the action are, the more will the company gain in relation to the cost from that certain action.

Table 5.4: Actions and their relevance values

Action	Relevance value
Get Sesam in use	3
Implement matrix	2.8
Implement time-control	2.7
Develop merging script for BTR-tool.	2.5
Convert old test cases	2.3
Automate version and standard dependent fixture selection.	1.8
Group tests according to functionality specification.	1.7
Structure fixtures/setups	1.5
Register suites automatically to X-unit	1.5

Even though some actions that obviously result in better changes than others and would seem to fit better higher up on the list of relevant actions, the cost greatly affect the worth of an action. If an action results in fairly low profit but is very cheap to perform, the action can be more worth than an action with good profit but very high cost. As can be viewed in table 5.3, the action ‘Implement matrix’ results in a much higher improvement value than does the action ‘Get Sesam in use’. However, since the cost of getting Sesam in use is almost nothing compared to implementing the matrix, the ‘Get Sesam in use’ action gets a higher relevance value than ‘Implement matrix’, as can be seen in table 5.4.

5.4 Improvement summary

The preceding chapter aimed at finding areas where improvements could be made and a uniform way of working could be achieved. A number of improvement ideas has been presented and rated. There are however some differences between Base and CPP development that makes it impossible to use the same way of working in all areas. Mainly, Base testing environment runs directly on Solaris platform whereas CPP runs upon SoftOSE real time OS. CPP also has external requirements towards Ericsson, which make CPP less controllable.

Testing on Base and testing on CPP also differ because they are intended to test different functionality. Many areas are however already very much alike as it is now. Regarding the make system, Base and CPP have a similar structure. Both have a module-specific make file and a general make file for the different build goals, such as PureCoverage and Purify. Both have gone from using script languages and FrameMaker documents to using X-unit, which has enabled both developments to have their testers write program code and test code in the same languages. Both structure their test cases into suites, which is not strange since it is the intention of X-unit. Both use SQR (Software Quality Rank) to estimate how well testing is being performed in order to reach a higher code quality. This implicates that both testing departments also uses Quantify, FlexeLint, PureCoverage and Purify to measure how well testing is going. Both have moved advanced functionality away from the test cases so that writing the actual test cases require few lines of code. Both departments has there by acknowledged that testers needs to be able to lay focus on the specific test cases, instead of being held up by technical preparations and such. Both have moved from test cases having other test cases as pre requisite to test cases being stand-alone. This was a great improvement to the testing environments.

Following the aspects discussed in chapter 5 will lead to a more common way of working than there is today and also lead to a number of improvements to the testing. If Base converts all old test cases, as have been done on CPP, the configuration files can be removed. This will remove an unnecessary intermediate step, which is only performed in order to satisfy old test cases. If Base and CPP change the way they work with setups/fixtures, more can be reused and automated, which also lets testers focus more on the test cases. The action also results in a more common way of working between the departments. If CPP begins to work towards the use of a matrix, they will get a better overview of functionality coverage, as Base has today. A restructuring of test cases into matrices according to the SoC would be an improvement on Base and the same approach should be taken by CPP when structuring their test cases into matrices. CPP can remove the time flow in their test as have been done on Base, which will ease the work for testers as well as result in a more common way of working.

Regarding BTR-tool, if the tool is developed so it works well on Base, CPP can create an equivalent tool so that both will have a uniform way of creating test reports. Improvements made to the tool can henceforth also be exchanged between the departments.

Sesam should be brought into use on Base to see how well it improves testing. If Sesam proves to work well, a similar tool can be developed for CPP, which would be another step towards a common way of working. Bringing Sesam into the daily testing will also give the testing environments a good debugging tool.

6 Summary of the interviews

This chapter summarizes from which persons the different facts has been acquired and also their respective position at the company.

6.1 Base interviews

The facts presented in the essay, regarding Base in general and their area of production (see chapter 3.1) is taken from interviews with Fredrik Kastengren at TietoEnator. This person is a well-educated employee and knows well what he is working with. His task was to give a general description of what Base is and does on a high level.

The information regarding Base testing environment has been taken mainly from interviews with five persons; Benny Saxén, Jonas Arvidsson, Daniel Nordkvist, Johan Torstensson and Peter Heubeck.

Benny Saxén has mainly provided information about Black Box library, the configuration files, message timers, standards and versions, the test case matrix and all the different tools. The tools were Sesam, CNF-builder and BTR-tool. (See chapter 4.1). He also showed us the difference between writing test cases today between writing test cases before Black Box library. Benny presented most of the facts on a fairly high level in order to give a good picture of Base since that was what we needed when coming to the company as newcomers. Benny is one of the head persons at Base and has been involved in developing parts of Black Box library and Sesam as well as some other tools.

Jonas Arvidson is the leading developer of Black box library, Sesam and some other tools. He is a highly respected employee and known for his carefulness and knowledge. He has mainly contributed with developing the requirement specification, judging of our progress, technical details about Black box library and the tools on Base. He has also confirmed the information acquired from other interviewed employees at Base.

Daniel Nordkvist provided us with some technical details about Base. He talked about the test case matrix on a more detailed level than Benny Saxén did (see chapter 4.1.4) and gave some code examples. He also showed us some examples of test cases on Base.

Johan Torstensson mainly gave us information about BTR-tool (see chapter 4.1.5.3), the automation tool on Base for creating test reports. Johan Torstensson is one of the developers of BTR-tool so he has detailed knowledge of the tool. He presented what the tool is used for and what the current liabilities of the tool are today. Together with him we also discussed some ideas to improve the tool. This was necessary since we would not have been able yourselves to say for sure that a certain idea would be able to perform. The interviews with Johan also brought up the ideas with structuring test cases according to a function specification, called the SoC (Statement of Compliance) on Base.

Peter Heubeck also provided us with information about BTR-tool and we also discussed improvement possibilities with him. Peter is the main developer of BTR-tool. It was good to discuss BTR-tool with both these developers to be able to be more certain that the improvement ideas were possible to perform. Since both these persons have knowledge on code level, they can easily tell what can and what cannot be achieved.

6.2 CPP interviews

The facts presented in the essay, regarding CPP in general and their area of production is (see chapter 3.2) taken from interviews with Fredrik Kastengren at TietoEnator. This person is a well-educated employee and knows well what he is working with.

The information regarding CPP testing environment has mainly been provided by Jonas Arvidsson, Greger Olsson, Anders Eriksson, …

Anders Eriksson has mainly contributed with how the old testing environment worked, with the usage of scripts and he was the person which created the first version of the requirement specification. He has also presented information about TietoEnator itself and what they do and also a bit about the structure of the make system. Today Anders Eriksson works at CPP and has been an employee at the company for many years so he has good knowledge about the company.

Greger Olsson has given information about the structure of CPP's testing environment, the fixtures, usage of X-Unit and test case creation (see chapter 4.2). Greger is one of the front figures of CPP and has good knowledge of CPP testing environment. Together with Greger we also discussed a lot of the improvement ideas to see if they were of true interest and if they were possible to push through.

Jonas Arvidsson has also provided a lot of information regarding CPP. He is a former Base worker but is now working at the CPP department. Jonas brought up information about X-Unit on a technical level; how it is incorporated in the CPP testing environment and also what prompted the usage of X-Unit. He has confirmed the information acquired from other interviewed employees at CPP, as well as with the Base interviews.

The persons listed are, as mentioned, the main persons interviewed, which have contributed with the largest parts of the essay. Other persons have also been interviewed, but their contribution to the essay is small or even in some cases non-existent.

7 Recommendations

If we have had more time, we would have tried to dig deeper within the different areas to try and find out other possible improvements and also more details around the improvements brought up. The make system could have been investigated more through roughly to find out if any restructuring can be made to improve build time. This was something we started on at the early stages of the investigation but we felt that we would not have the time to perform any usable results in this area. We are certain that looking into the make system and the build structure of the testing environments could make enough work to work as a complete separate D-dissertation. The are much technical details here to investigate and the project would surely involve a lot of test running and analysis's of results regarding differences in execution times using different approaches of builds.

Regarding BTR-tool, we could have begun to work on the scripts mentioned ourselves, with a little help from the staff. For someone to continue on the work we have performed, this is one of the most obvious actions to perform.

We would also have wanted to interview even more people from the staff to find out what they feel about their testing environment. Since different people have different views, we are sure that some more issues would have come up in addition to the ones that came up from the interviews we did. The most interviews we performed were also with well-informed head persons which daily work mostly consist of project management, testing management and other leading roles. Having interviewed more of the testers would have proved more if the picture that the leading persons have actually match the pictures of the persons whose daily work is the actual testing process.

Most importantly, we would have liked to have the time to experience the testing environments ourselves to get a feeling of how much effort and time is required to create and run tests on the different developments, to see if there are any major differences. We already knew from the start that we would be able to spend much time on digging into code and getting a thorough feeling for the environments ourselves but mainly would have to rely on interviews and demonstrations. The problem with doing this is that people, which have worked with the same assignment and environment for a long time tend to accept and hold the way of working, which they have learnt, very dear. Generally, people are not happy with changes. If we, as newcomers, had looked deeper into the environments, we would probably have found matters that did not come up during interviews, since the long-time employed personnel overlook them.

If we would redo the entire project, we would have focused more on the requirement analysis in the beginning of the project. The requirement specification we acquired from TietoEnator was vague and unspecified. Naturally we brought this matter up for discussion and saw to that the specification became more clear and specific. Since we are not experienced developers we could not exactly know how much we could expect from the specification. After a couple of revisions we accepted the specification and began the investigation. Further on, it came to evidence that the specification still was too unspecified and unclear, so we had to move back to square one and bring forth further clarification of the specification. We had to do this a number of times before we felt that the work we needed to perform were clear enough to us and that the goals at hand were achievable to us during the time we had. The reason for the bad communication between us and the persons responsible for the D-dissertation assignment, as we realized further on, was the great gap of knowledge difference between us. Whereas the specification was clear enough for them, which had worked within the testing departments for years, was very unclear to us, which had hardly any experience of testing at all. Redoing this step we would have trusted our senses more and seen through that the specification was really clear enough to us before beginning the actual work. If we had done so, we would have saved us a lot of time and unnecessary work. We had a hard struggle trying to realize and extract the essentials of the specification and this could have been avoided by actually trust ourselves when feeling that the information we got to work with was not enough. The first and the last revision versions of the requirement specification from TietoEnator are included in the essay as appendixes to show the differences between them. Even though the difference does not seem that big, it took us a lot of time to bring forth this further clarification. However, redoing the requirement analysis today, we would have seen to that the last revision of the specification had been even more specified. Even though we had heard many times how important it is to spend a great amount of time on requirement analysis we felt the need to begin the actual work. This is where we made our biggest mistake in this project.

Basing a research mainly of interviews requires some precautions and has some complications. One should not perform an interview and then just blindly trust the information that has been provided by the interviewed person. What is acquired from a single interview is the view of that single person, which may or may not correspond to reality. For such a research to be trustworthy in any sense, a number of people need to be interviewed, to find out if the views of the persons do match. Furthermore, performing several interviews regarding a certain matter will give different perspectives of that particular matter. One thing that looks good at first sight may actually be worthless when looking at other perspectives.

8 Summary of the project

Before we began working on this project we had many thoughts about what actually should be done. We had gotten an assignment, which should aim at improving testing strategies on a company with personnel that have worked with this for over 15 years. It felt like it would be hard to come up with improvements when having no background at all and so little time to learn about the system. Many improvements in this area has also already been done recently.

The project began with analyzing the requirement specification gotten from TietoEnator. The goals of the assignment were very unclear at the beginning so much effort were required to find out what the company actually wanted to be done. Truth is that the ones responsible for the project did not have a clear picture of what should be achieved. What they knew was that there were some areas that could use improvements and that they wanted a more common way of working, hence the unspecified assignment. This resulted in us having to lay hard work in finding out what to aim for. The requirement analysis was also difficult to perform at the beginning because we did not know much about the testing environments. During the project we interviewed a lot of personnel on both Base development and CPP development. Since the persons interviewed had detailed knowledge in their respective areas it was hard to understand everything that came up. What helped a lot was that we recorded most interviews, which gave us the possibility to analyze details afterwards. As we interviewed personnel, we received more knowledge about the testing system and could thereby come up with better questions regarding the requirements of the project. Every once in a while we had a discussion with the person responsible for our project to discuss our ideas and his expectations. When we decided upon ideas to improvements, we discussed them together with persons having detailed knowledge in the different areas to find out if our ideas would be useful and possible to apply.

The personnel on TietoEnator were very helpful and they gladly took time to help us whenever they had time to do so. The interviews therefore went very smoothly.

Through this research the company gained knowledge about some possible improvements and some ideas on how to get a more common way of working regarding testing on Base and CPP. Besides that, they also got proof that because the differences between the developments, it is not possible to work in an entirely uniform way.

References

[1] Kent Beck. Extreme programming explained: embrace change. 20(1):61641-6, 1999.

[2] Why test software, http://www.testline.co.uk/testing_whytest.htm, 2005-02-28

[3] Testing, http://www.stctek.com/faq.html, 2005-02-28

[4] Software Testing, http://sern.ucalgary.ca/~sdyck/courses/seng621/webdoc.html, 2005-02-29

[5] Testing tools, http://www.cs.queensu.ca/home/shepard/testing.dir/under.construction/tool_list.html, 2005-03-10

[6] X-unit, http://encyclopedia.thefreedictionary.com/Xunit, 2005-03-10

[7] Erik Meade, http://c2.com/cgi/wiki?ErikMeade, 2005-05-05

[8] Unit/Component testing, http://www.developer.com/java/other/article.php/2217461, 2005-05-11

[9] Component testing, http://www.cs.clemson.edu/~johnmc/joop/col3/column3.html, 2005-05-11

[10] Grey Box testing, http://www.geocities.com/xtremetesting/GreyBox.html, 2005-11-28

[11] Rolf Ejvegård, Vetenskaplig metod, Andra upplagan.

[12] Software testing, http://www.centipedia.com/articles/Software_testing, 2005-12-01

[13] Davis, A. 201 principles of software development, McGraw-Hill, 1995.

[14] Hutcheson, Marnie L. Software testing fundamentals. Wiley, 2003.

[15] Binder, Robert V. Testing object-oriented systems – models, patterns and tools. Addison-Wesley, 2000.

[16] Kaner, Cem, James Bach and Bret Pettichort. Lessons learned in software testing. Wiley 2002.

[17] Patton, Ron. Software testing. Sams publishing, 2001.

[18] McGregor, John D and David A. Sykes. A practical guide to testing object-oriented software. Addison-Wesley, 2001.

[19] Sommerville, Ian. Software Engineering. Addison-Wesley, 6th edition, 2001.

[20] Perry, William. Effective Methods for Software testing. John Wiley & Sons, 1995.

A Requirement Specifications

A.1 First Version

Below is the ***first version*** *of the requirement specification received from TietoEnator.*

Revision history

Date	Rev	Description	Prepared
2005-02-14	0.1	First draft	Anders Eriksson
2005-02-17	0.2	Updated after comments	Anders Eriksson

Terminology and Definitions

The following definitions and explanations of certain words and abbreviations are applicable within this document:

Base development	Design organisation developing software common to more than one platform
BB	Black box
Code coverage	A value of the amount of code that has been executed during a test phase
CPP	Connectivity Packet Platform – one of Ericsson's telecom nodes
CT	Component Test – the first level of test
Requirements	External expectations on something
Sub business unit (SBU)	Organizational unit at the 2nd level, consisting of a set of Units

Background

Base development is today using a black box (BB) oriented approach towards component testing (CT). There is a black box library (BB-lib) written in C++ used to configure and set up different prerequisites before the test cases are executed. The lib is also used to perform some of the test cases and to verify that the tested code behaves as expected. CPP have implemented both white box and black box test cases. In both environments the test cases are written in C++.

Goals

- **Achieve a common way of working regarding CT for both base development and CPP development.**
- **Describe the way of working in a guideline document.**

The requirements on CT's are:

- **High code coverage**
- **Fast execution**
- **Easy to write new test cases**
- **Simple maintenance**

A.2 Last Version

Below is the ***last version*** *of the requirement specification.*

Revision history

Date	Rev	Description	Prepared
2005-02-14	0.1	First draft	Anders Eriksson
2005-02-17	0.2	Updated after comments	Anders Eriksson
2005-02-22	0.3	Updated title and chapter 3.1 background	Anders Eriksson
2005-03-07	0.4	Updated chapter 3.2 goals	Jonas Arvidsson

Terminology and Definitions

The following definitions and explanations of certain words and abbreviations are applicable within this document:

Base development	Design organisation developing software common to more than one platform
BB	Black box
Code coverage	A value of the amount of code that has been executed during a test phase
CPP	Connectivity Packet Platform – one of Ericsson's telecom nodes
CT	Component Test – the first level of test
Requirements	External expectations on something
Sub business unit (SBU)	Organizational unit at the 2nd level, consisting of a set of Units

Background

Component Test is the first test phase. It is written and executed by the software designers writing the code to be tested. Historically, the Signaling Base Development department has used a proprietary tool called Module Test Tool (MTT) for this purpose. While possessing many of the desirable characteristics of a component test environment (ease of use, ability to automate the running of large test suites, etc.), MTT had some drawbacks that prompted the development of a new component test strategy based on the X-Unit framework some years ago. The main problem that needed to be solved was that the turn-around time for running a test, finding a fault, correcting it, and running the test again was too long.

The first step was to automatically convert all the MTT test cases to X-Unit test cases in the C language. The test framework was then extended with so-called stubs that simulate the execution environment in which the tested code normally runs. The next step, which was taken for some of the more complex signaling modules, was to extract common functionality used by many test cases into a library called BB-lib, Black Box Library. Test cases written for execution in BB-lib are smaller and less complex than regular X-Unit test cases, since much of the module set-up and message verification has been moved into the library.

For CPP5, a decision was made to also move to X-Unit based testing. Prior to CPP5 all component testing was based on a platform called SimCello, which is basically comprised of a set of configurable simulators representing all the subsystems in CPP. SimCello suffers from several drawbacks from a component testing point of view, as described in the sections below.

The SimCello platform is built into one large binary containing all the simulators and the test object. Just to build and link this binary takes a lot of time on anything but really high-end machines. Further, since a lot of ClearCase VOB elements are involved, this also increases the build time. This does not fit well with component testing where building and executing tests should be as quick as possible.

Tests are written in the TCL script language, which stimulates a test driver written in C, which in turn does something with the test object. The TCL script then looks in a log to see if the test object logged what was expected. This is less than optimal for several reasons; designers have to write tests in a language which is not the same as the test object is written in; there is a level of indirection between the tests (TCL scripts) and the test object which is the C driver (which needs to be updated when the test object is updated); the tests check that the test object logs what is expected, not that it does what is expected.

There are dependencies both towards the simulators, which are developed by other subsystems, and towards the SimCello build support, which is quite complex. Interface or behavioural changes in subsystems have to be delivered before they may be tested by other subsystems.

White box testing is pretty much impossible. Without modifying the intermediary test driver, there is no way to test a specific section of code. The designer have to write test cases that send signals to the test object which, hopefully, will result in the intended section of code to execute. This makes it very hard to get coverage of some parts of the code.

The move to X-Unit based testing in CPP started off with creating simple stubs of the external interfaces used by the SS7-related modules developed in Karlstad. These stubs replace some of the simulators earlier found in SimCello. Then, the various teams started to build test cases for the SS7 components. In the beginning there were no direct guidelines available for this, which resulted in the component tests being somewhat dissimilar. Except for the stubs, which are shared by the component tests, there is no glue code or library (like BB-Lib in Signaling Base Development); CPP uses the Check X-Unit library directly.

Goals

Propose a common way of working regarding CT for both Base development and CPP development and locate areas where improvements can be made. The requirements on CT are:

- High code coverage
- Fast and simple execution
- Easy to write new test cases
- Simple maintenance
- Error detection
- Functionality coverage
- Automation

Describe the way of working in a guideline document. The description shall include details such as ideas to solve issues regarding automatic reporting of test results with BTR-tool and the needs and possibilities to use BB-lib on CPP.