Unit Test Generator (UTG)

Requirements

Revision AP1
July 23, 1999
File:

TELLABS

1 Introduction

The Unit Test Generator is a tool which aids in the creation of unit test scripts from **C** source code.

1.1 Purpose, Scope and Field of Application

1.1.1 Purpose

The purpose of this document is to identify all of the requierments necessary to design, implement and test the **UTG.**

1.1.2 Scope

This document covers the unit testing phase of the TITAN©532L software lifecycle.

1.1.3 Field of Application

This document is written for Tellabs Research and Development Groups and the Purdue CS406 and CS407 classes

1.2 Revision History

Revision	**Date**	**Revision Description**
AP1	**7/21/99**	**Initial Version**

1.3 References and Attachments

- TITAN©5300 Software Unit Test Guidelines
- TITAN©5300 Software Test Script Standards
- Sample source code and unit tests are available

1.4 Definitions, Acronyms and Abbreviations

1.4.1 Definitions

Term	Definition
TITAN©532L	A DS 1 / DS 0 level digital cross connect system from Tellabs
Unit Test Script	A Unit Test Script is a text document written in a mix of english and pseudo code which outline how to test a single function

Table 1: Definition Table

1.4.2 Acronyms and Abbreviations

Acronym/ Abbreviation	Meaning
UT	Individual test in a unit test script
UTG	Unit Test Generator
GUI	Graphical User Interface

Table 2: Acronyms and Abbreviations

2 Requirements Overview

When dealing with system requirements it is important to remember several key concepts:

- Requirements are driven by the product necessarily not vice versa.
- Requirements change when it most inconvenient.
- There are few wrong answers but some answers are more right than others.
- Requirements are quite often a wish list.

3 UTG Introduction

One of the many facts of life in the realm of software development is that no code (of reasonable complexity) will ever work perfectly on the first try. For this reason, testing is an integral part of any well defined software lifecycle.

Any company which is serious about software development, especially on mission critical systems, will have a well defined process in place describing exactly how testing shall be performed. At Tellabs, on the TITAN©532L, project the first testing phase is the unit test phase. During this phase the function is the entity which is tested. Every function written for the project is tested as independently as possible from the rest of the software. This testing tends to bring out simple coding mistakes such as memory leaks, array boundary errors, error codes not being properly handled and resources which are locked but not released. It is much more difficult to find these faults when a function is running live in a system.

To ensure that unit tests are performed (and performed correctly) unit test scripts are created by the author of a function. The script is written in a mix of pseudo code and english. Since there are no good ways to automate function level testing, tools such as ObjectCenter© must be employed to execute the tests by hand. It sometimes happens that bugs can be found in the code simply by creating the test script itself.

The process for unit testing calls for each line of code in a function to be tested. Block level testing lends itself well to this requirement. Most individual tests in a script will cover a single block of code. The script should contain enough tests to guarantee that every line of code (or block) in a function is executed[1].

The unit test script itself, is comprised of several different sections. The first section is the file head-

1. Keep in mind that this is *not* path coverage testing so tests do not necessarily need to be created to cover multiple code blocks.

er and contains items such as the file name, version and purpose. The next section contains the summary of all the tests in the script (a one line description of a test). The last section contains each of the tests.

Each test has several sections to it as well. The first section of the test gives the test number and test title. There is a section which outlines the inputs of function parameters and local variables required to execute the block. The next portion of the test is for expected outputs. These include a list of functions (with return codes) which are called after the block has been executed and the return code of the function which is being tested.

As can be imagined, creating these unit test scripts are a very tedious and time consuming process. Any tool which can simplify and automate (even portions of) this process would be very valuable indeed. The goal of this project is to create such a tool, the Unit Test Generator (UTG).

4 UTG Core Functionality

In order to make the UTG a useful tool, several features must be implemented. The first and most feature of the UTG is its ability to break down any valid **C** function into its individual code blocks.

The next major feature of the UTG is its ability to manipulate entire tests in the script. The tool should have the ability to perform most editing features on entire tests (i.e., cut, paste, insert, delete, etc...).

Another feature of the UTG will be the ability to provide default values for tests. This is especially useful in the Input and Output sections of the tests. To take advantage of this feature, the user would enter in the text to be used as default and would then choose which tests would be seeded with the default values.

The final major feature of the UTG is its ability to provide a guess as to what the title of the test should be. For instance if a block of code was begun as: **IF (FUNC() != 0) { ... }** then the UTG would guess that the title would be, **FUNC() == 0**. It is not expected that the UTG be %100 correct when suggesting a title, the tool should simply negate the condition which created the block and use that. No run time analysis will be attempted.

5 UTG Interface

While there are no steadfast requirements governing the user interface for the UTG, it is required that some form of GUI be used. The GUI should be intuitive, easy to use and provide access to all of the features of the UTG.

6 UTG General Requirements

• The UTG shall default to the following naming scheme when dealing with source code file names, function names and unit test script file names: The source code filename shall be the function name with the *.c* suffix and the unit test script file name will be the function name with the *.ut* function name. For example the function **func1()** will have the filename **func1.c** and the unit test script file name, **func1.ut**.

• As an added feature, the UTG will be able to take a **C** function, break it down into its individual code blocks and display it back to the user highlighting the different blocks.

• The UTG can assume that functions used as input are free from compile errors.

• The editing features supported by the UTG will be, insert, delete, edit, copy, paste and move. The user should be able to perform these operations on entire tests, not just portions of text.

- Each test in the script is numbered, the tool will assign the number when the unit test script is generated.
- The UTG must be able to read in existing unit test scripts. The tool does not necessarily have to be able to read all existing test scripts, poorly formed test scripts can be ignored.
- The code block(s) and line numbers from the source code must be included in each unit test.

7 System and Environmental Requirements

7.1 Platform Requirements

- The UTG must be compilable and executable on the Solaris (version 2.6) operating system.

7.2 Source Language Requirements

- The UTG may be developed using any freely available tools and languages which are currently installed on the Tellabs Intranet or can be installed relatively painlessly. While almost any language may be used, **C** and **Perl 5** are highly recommended since those are already widely supported at Tellabs.

7.3 Timing and Speed Characteristics

- While there are no specific speed or timing requirements for the UTG, the tool should run in a *reasonable* amount of time.