Standard

Unit Test Generator (UTG) | Requirements | B | 1/13/2000
Subject | File Code | Rev. | Effective Date

Rev	Change Description	Date - Orig
A	Initial Version	7/21/99
B	Incorporate changes from CS406 class	1/13/99

1. Introduction

Introduction

The Unit Test Generator is a tool which helps software developers create and manage unit test scripts.

Purpose

The purpose of this document is to outline all of the requirements for the UTG. These requirements will be used to design, implement, test and deliver a working incarnation of the UTG.

Scope

This document covers the requirements phase of the UTG's lifecycle.

Field of Application

This document is written for the Tellabs software development groups, Dr. Aditya Mathur and the students of the CS407 class at Purdue University.

Revision History

Revision	Date	Revision Description
A	7/21/99	Initial Version
B	1/13/99	Incorporate any and all changes from the CS406 class.

Definitions

Term	Definition
clipboard	An area of memory where unit tests can be copied to and its value will be maintained until a new item is copied to the clipboard or the program is exited.
craft	This is a commonly used name to refer to a person who configures, provisions and maintains Tellabs and other equipment.
code block	A code block is an atomic piece of software which has exactly one entrance point and one exit point.

Acronyms and Abbreviations

Term	Definition
SWIT	Software Integration Testing (Feature Level Testing)
RSIT	Regression Software Testing (System Level Testing)
UT	Unit Testing (Function Level Testing)
UTS	A Unit Test Script

Term	Definition
UTG	The Unit Test Generator

Requirements Overview

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

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

2. The Unit Test Generator

Introduction

One of the many facts of life in software development is that no code of reasonable complexity will ever work perfectly the first time. For this reason testing is an imperative and required step in any mature software lifecycle.

The TITAN 532L software development group has three separate testing phases. Each phase is designed to test a different layer of software. The first phase of testing is unit testing (UT). Its sole purpose is to test at the function level. Next comes the software integration phase (SWIT). During this phase, tests are designed to isolate and test a single feature. The final phase, regression software integration testing (RSIT), is used to test the entire system.

Each set of tests has its own purpose. The UT phase hopes to find resource allocation problems, memory leaks, array boundary overflows and other minor coding errors.

SWIT tests are designed around individual features. They may be executed to test a new craft command, piece of hardware or a complex software algorithm. This phase attempts to catch problems with the interactions between functions in a single feature or sub-system. This testing is critical when testing the interface between software features, various hardware components and the firmware which may control them.

RSIT tests are employed to test the entire system. It is necessary to make sure that individual features work well together. Deadlock and starvation scenarios are most often found at this level.

Unit Test Plan

Creating a test plan at any level is a difficult and time consuming process but by far the most tedious is the unit test plan.

Every function which is written for the 532L is required to have a unit test plan. This plan must *thoroughly* test the code it was designed for. While it would be ideal to state that a unit test plan must *completely* exercise a function it is not feasible. This would require that each and every possible execution path of a function be tested. This is not possible for any large piece of software.

Instead, each unit test plan must outline a means of making sure that every line of code is executed. A more practical way of stating this is that every code block must be executed at least one time during the course of the unit test.

Code Block

While there is some disagreement on what a code block actually is, for the purposes of the UTG, a code block is an atomic group of statements which have exactly one entry point and exactly one exit point. Code blocks cannot be nested. A function call in and of itself does not define a code block (but it is contained in a code block).

3. Unit Test Script File

File Format

Listed below are the different sections required in a Unit Test script. They are listed in order of appearance.

File Header
This section is always the beginning of the file

```
/************************ BEGIN ************************

*       COPYRIGHT %G% %U% BY TELLABS OPERATIONS, INC.
*                     ALL RIGHTS RESERVED
*           THIS DOCUMENT MAY NOT BE REPRODUCED WITHOUT
*               THE WRITTEN PERMISSION OF TELLABS
*       4951 INDIANA AVENUE, LISLE, IL. 60532
*
*****************************************************

*
* Name:          <filename>
* Type:          Unit Test Script
* Originator:    <Author>
* File:          %M%
* Version:       %I%
* Reference:     <Reference Identifier>
* Description:
*      Description
************************ END *************************/
```

Block Definitions
This section is optional

```
Block Definitions:
==================
<Block number>: <start row> <start col> <end row> <end col>

------------------------------------------------------------
```

Summary:
This section is automatically generated by the UTG

```
Summary:
========
1)
2.1)
2.2)
------------------------------------------------------------
```

Optional Section
This section can contain any data. The data here is only an example.

```
Default Input:
==============

------------------------------------------------------------
```

Continued on next page

3. Unit Test Script File (cont.)

Unit Tests

There will be from 1 to N unit tests in a unit test script. The test number reflects the two different numbering schemes (See Req. 7-9 on page 12).

Test [X | X.Y])
Summary:
<*Test summary*> - This will appear in the summary section.

Blocks: 64,65

Input:
<*input*>
Expected Output:
<*output*>
Status:
<Pass or Fail>

Footer

This must be the last section in the unit test script.

```
==========================================================
Date:
Signature:
----------------------------------------------------------
```

4. UTG Core Functionality

Functional Requirements

Req. 4-1	The UTG must make the unit testing phase easier, more consistent and more robust.
Req. 4-2	The UTG must be able to generate a complete list of code blocks from all valid **C** files.
Req. 4-3	Given a **C** file, the UTG must be able to create a unit test script template.
Req. 4-4	The UTG must be able to read in existing unit test scripts which were previously generated by the UTG.
Req. 4-5	The UTG should also be able to read most unit test scripts which were not created by the UTG.
Req. 4-6	Given a UT and **C** file, the UTG must be able to determine if a unit test script is out of date with the corresponding **C** file.
Req. 4-7	If a UT file and **C** file are out of sync then the UTG must inform the user. The user may choose to ignore this condition otherwise the user should provide a means of synchronizing the unit test and source code.
Req. 4-8	A Graphical User Interface is required to control the UTG.

Legacy Scripts

Req. 4-5

Since a large number of unit test scripts already exist, the UTG must be able to read in these scripts as well. These were written in a wide variety of formats and with different levels of quality, so the UTG will not have to be able to read in all existing UTs. The UT reader can key off of the file format provided (See “Unit Test Script File” on page 5.).

If an existing UT script cannot be parsed by the UTG then the user must be notified so that the file can be reformatted. The user should be told which line(s) was affected as well as the section it was trying to read it as.

5. Source Code Requirements

Source Code Requirements

Req. 5-1	Use of CPP to include header files, perform macro substitution and evaluate pre-processor conditionals is not required.
Req. 5-2	The UTG must be able to parse both K&R **C** and ANSI **C** files.
Req. 5-3	If the UTG is unable to create a code block list due to compile errors, the error must be returned to the user.
Req. 5-4	The user must be able to edit a unit test script if the corresponding **C** file is not currently loaded or if it contains errors.
Req. 5-5	The UTG must generate statistics on code block usage inside a unit test script. Total block usage and individual block usage statistics must be kept.

Preprocessor

Req. 5-1

While it would be more correct to compile the code with CPP, this will not be required. It should be possible for the UTG to generate a code block list without the pre-processor. In the minority of cases where CPP is required to generate the code block list, an error may generated and displayed to the user.

5. Source Code Requirements (cont.)

Code Block Statistics

When the user is finished creating or editing a unit test script, it should be possible to generate a code block usage report. This report will show the percentage of total blocks used in the UT script as well as number of times each block is used in the script. For instance, assume that there are 10 total code blocks. Block 1 is used 1 time, block 2 is used 2 times, block 3 and 4 are never used, and blocks 5 -10 are used a total of 4 times. The report may look something like:

Table 5-1 Code Block Usage Report

Block Number	Usage
1	1
2	2
3	0
4	0
5	4
6	4
7	4
8	4
9	4
10	4
Total Code Block Usage:	**%80**

The above table layout is not necessarily the layout which should be used. The report is required to have the following information: the source code filename and path, the unit test filename and path, the current time and date and a table similar to the one listed above. The user should have the ability to save this report to a file.

6. Graphical User Interface

GUI Functionality

Req. 6-1 A graphical user interface must be provided with the UTG.

Req. 6-2 The graphical user interface must provide the following functionality:

- **Open** an existing unit test or **C** file.
- Generate a **New** unit test script
- **Save** a unit test script under the current filename if one exists. If there is not currently a filename, the GUI must prompt the user for a name.
- **Save** the current file **as** a different file.
- **Quit** the UTG

Req. 6-3 The GUI must provide a text widget to display the entire contents of **C** file.

Req. 6-4 The GUI must provide a text widget to display the entire contents of a UT file.

Req. 6-5 A Code Block Widget must be provided to display the contents of a **C** file by code block.

Req. 6-6 A Unit Test Widget must be provided to display the contents of a UT file by unit test.

Req. 6-7 The GUI must also provide the user with the standard text editing capabilities: cut, copy, paste.

Code Block Widget

Req. 6-5

The code block widget will only display a list of code block definitions. Each code block should be displayed on its own line and take the form:

B *<block num>***: (***<start row> <start col>***),** ***(****<end row> <end col>****)***

If the user clicks on a block definition in this widget and a **C** text widget is active the text widget should warp to the beginning of the block definition. The first line of the code block should be centered in the widget.

Continued on next page

6. Graphical User Interface (cont.)

Unit Test Widget

Req. 6-6

The unit test widget will only display the number and title of each unit test found in a unit test script. Each unit test should be displayed on its own line and take the form:

UT *<UT num>***:** *Unit Test TItle*

If the user clicks on a unit test in this widget and a UT text widget is active, the text widget should warp to the beginning of the unit test. The first line of the test should be centered in the widget.

Text Editing

Req. 6-7

In addition to the test editing features (See section 7. “Editing Requirements”.) provided by the UTG, it must also provide an easy means of performing the standard text editing commands, cut, copy and paste.

7. Editing Requirements

Editing

Req. 7-1	The UTG shall provide the ability to insert a new test into an existing test script.
Req. 7-2	The UTG shall support the ability to delete a test from an existing test script.
Req. 7-3	The ability to copy a test shall be supported.
Req. 7-4	The ability to cut a test from a script shall be allowed.
Req. 7-5	The UTG shall provide the mechanism to paste a test which was either cut or copied into an existing script.
Req. 7-6	The UTG shall provide the ability to undo one high level editing operation.
Obj. 7-7	The UTG should be able to undo up to 10 high level editing operations.
Req. 7-8	The UTG is responsible for numbering all tests.
Req. 7-9	Two levels of numbering are supported by the UTG.
Req. 7-10	The numbering scheme of a test can be changed by the user at any time.
Req. 7-11	The Summary section of a unit test script is automatically generated by the UTG.
Req. 7-12	Three default I/O modes are supported by the UTG. Auto update mode, manual update mode and no update mode.

Test Level Editing

Req. 7-1
Req. 7-2
Req. 7-3
Req. 7-4
Req. 7-5
Req. 7-6
Obj. 7-7

The UTG will supply several high level editing features so that users are able to create and edit unit tests with greater ease. When the user decides to inset a new test, a unit test template should be inserted where at the current cursor position. This means that a new test can be inserted between two existing tests, at the beginning of a script or at the end of a script. In addition to being able to insert a new test, the user should be able to delete an existing test.

If the user decides to either cut or copy a test from a script then that test should be copied into the clipboard. Copying a test is a non-destructive operation. Cutting a test from a script copies it to the clipboard and then deletes it.

The user may also paste a test into a script. This operation should behave the same as the insert operation with the exception that the test is taken from the clipboard.

Continued on next page

7. Editing Requirements (cont.)

Test Numbering

Req. 7-8
Req. 7-9
Req. 7-10

The UTG supports two numbering schemes. The first one sets the test numbers as 1, 2, 3, etc.... The second numbering scheme sets the test number as 1.1, 1.2, 1.3, etc.... The numbering scheme is determined on a per test basis. In other words the user can have a numbering scheme such as: 1, 2, 3.1, 3.2, 4, 5. It is not valid to have the test numbers 5, 5.1, 5.2. The numbering schemes cannot be mixed in this manor. The test numbers will be 5, 6.1, 6.2 in this case. The default mode is 1, 2, 3,...

The user must be able to change the numbering scheme at any time.

The UTG is entirely responsible for numbering tests. Any time that the user does something to alter the test numbering such as insert, delete or change the numbering scheme of a test, the UTG must renumber all of the tests so that they maintain a sequential order.

Test Summary

Req. 7-11

The Summary section of the unit test script must be entirely created by the UTG. This section should be dynamically updated as the user is creating the script so that the summary reflects the current state of the script. It is only necessary to update the summary when the user finishes adding or changing the title. Not every time a new character is typed.

Default I/O Modes

Req. 7-12

When looking at a unit test script, one may notice that the Inputs and Outputs section of a unit test does not change very much between tests. When creating a new test it is often useful to have the Input and Output values default to those of the previous test. The UTG shall have 3 modes to support this feature. They are 1) Auto Update, 2) Manual Update and 3) Disable.

In auto update mode, the default I/O values will change with every test. This means that when I start test 2, the I/O from test 1 are given to me. When I start test 3, the values from test 2 are given to me. When I start test 4 I get test 3's I/O and so on...

In manual update mode, only update the defaults when the user clicks on this option. For example, I create test 1 and next test 2. There are no default I/O from test 1 because I didn't click on update defaults. Now I click to update defaults so when I write test 3 I get the defaults from test 2. When I write test 4, I get the defaults from test 2 and so on.

Mode 3 disables this feature. In this case there will not be any default I/O values.

8. Merging

Requirements

Req. 8-1 The UTG must be able to determine when the source code for a unit test script is out of date with the UT script.

Obj. 8-2 The UTG should inform the user if (and what) tests need to be modified, added or deleted.

Synchronization

Req. 8-1
Obj. 8-2

These requirements only deal with scenarios where the code block list from the unit test script and the source code file itself are available. If either of these things are not available then no merging needs to be done.

If the source file and code block list are available then the UTG must check to see if the UT and source file are synchronized. If they are not then the user must be notified. The user may choose to ignore this and continue editing the unit test script.

The UTG may provide the added intelligence of determining which code blocks have been changed, added or deleted from the source code since the UT was originally created. The user should be prompted with this information and provided with a means of updating the UT.

9. Environmental and Other Requirements

Miscellaneous

Req. 9-1 The UTG must be compileable and executable on Solaris Version 2.6.

Req. 9-2 The UTG must be written in languages which are freely available at Tellabs.

Languages

The UTG may be developed using any freely available tools and languages which are currently installed and supported at Tellabs (or can be installed relatively painlessly). While almost any language may be used, **C** and **Perl 5** are recommended as these are already widely supported.