tg

A Simple Test Driver Generator for Ada programs
Documentation for Version 3.1 of tg
Documentation revised and extended by Stefan Lucks, Feb. 2004

André Spiegel
Free University of Berlin
e-mail: spiegel@inf.fu-berlin.de
1 Introduction

A test driver is a program which tests a piece of software.

If you want to test something, for example an Ada package, you normally need to design a huge number of separate, elementary tests, which must be performed one after the other. Each elementary test typically consists of a subprogram call to some routine of the package, passing certain input data to it, plus a subsequent analysis of the results of that call. We call such an elementary test a test case.

It follows that, normally, test drivers are very simple but large programs. Mostly, they consist of endless repetitions of a single common pattern: that of a test case. Thus, it would be nice to have a tool that generates such driver programs automatically.

tg does just that. You feed it with a description of a test, given in some special, convenient format in which you only need to specify the essentials of each test case. tg takes this description – we call it the test script – and translates it into the source code of a corresponding test driver, which you can then compile, link, and execute.

tg’s native language is Ada, but you might also use it to test software written in other programming languages. (This would require ‘interfacing’ that software to Ada, since the test driver is always an Ada program.) tg itself is also written in Ada.

(Throughout this document, Ada refers to “Ada 95”, not to the now-obsolete version of the language called “Ada 83”. You cannot use tg in an Ada 83 environment. Though you might contact me, the author, for an old version of tg written in Ada 83 – I still have it on disc.)

1.1 A Small Example

Suppose you want to test an Ada function which counts the number of ‘&’ characters within a given string.

    function Count_Ampersand (Str : in String) return Natural;

One of your test cases might be to call it with parameter “abc&&&abc” and verify that the result is three. The code for this could look as follows.

    Put ("Testing three ampersands in the middle... ");
    begin
        Count := Count_Ampersand ("abc&&&abc");
        if Count = 3 then
            Put_Line ("pass.");
        else
            Put_Line ("fail.");
        end if;
    exception
    when others =>
        Put_Line ("fail.");
    end;

That is a lot of code for a single test case. In tg notation, to specify this test case, you would just write:

    ***** Testing three ampersands in the middle...
    test Count := Count_Ampersand ("abc&&&abc");
    pass Count = 3

The tg command translates this into code similar to that shown above and puts it into a complete driver program.
1.2 Testing Terminology

As it was explained above, tg views a test as a sequence of test cases. The test is described, or specified, in a test script file. You write this script in a special macro-type language which is described in the main part of this manual. The piece of software which you test is called the test item. It may be a single subprogram, or a package, or a complete software system.

We say that a test is performed by executing the test driver program. Likewise, the individual test cases are performed when the code into which their description was translated is executed.

The crucial point of a test case is a subprogram call to the test item. We call this the test call. The driver might do some preparatory work before the test call, and after the call has returned, the driver analyzes its results.

That means, there are three types of results: the result of the test call, of the test case, and of the complete test as a whole.

- The result of the test call is what it returns to the test driver. This can be the return value of a function, or the values of any out parameters. But it also includes everything that can be determined as an effect of the test call, like the state of a global flag, or output data written to some file, etc.

The result of the test call further includes the program path taken after it, because after an Ada subprogram call execution may either continue normally or it may be aborted by the raising of an exception.

- The driver finds the result of the test case by comparing the result of the test call to the expected result. If they are equal, the test item is said to have passed the test case, otherwise it failed it. We also say the result of the test case is either pass or fail.

There is a third possibility, which occurs when something unexpected happens in a test case outside the test call. For example, you might want to test a function that returns the length of a list, and you first have to build an example list to pass to that function. If during building that list something goes wrong, the result of the test case is not ‘fail’, but error.

The result ‘error’ is more precisely defined as an exception propagating out of any part of a test case, apart from the test call.

- Finally, there is a total test result which is defined as pass if the test item passed all the test cases, and fail if it didn’t. The total test result is undefined if the result of any test case was ‘error’.
2 Test Scripts

The term test script is commonly used for a document that describes how to perform a test of something. In the context of tg the word has a more specific meaning: it means a complete, machine-processable description of a test driver program which performs the test. In such a script only the essentials of the driver have to be specified. The tg command is nothing but a compiler for test scripts; it translates them into correct Ada source code.

Writing test scripts instead of coding the driver by hand not only saves a lot of work (the generated program is typically ten times larger than the script), but also helps to construct tests in a uniform, standardized way.

There are two main sections in a test script: the global section and the test case section. As you can tell from the words, parameters for the whole test driver are set in the global section, while the individual test cases form the test case section.

2.1 Test Script Basics

Test Scripts should have filenames ending with ‘.ts’.

A test script mainly contains chunks of Ada code, marked by keywords which tell tg at what place in the driver it should put that code. The basic idea of the format is to have the keywords begin in column 1, followed by the corresponding chunk of code, which can stretch over an arbitrary number of lines, all but the first of which must be indented. The first line which begins with a non-blank character marks the end of the chunk. (Empty lines, or lines containing only whitespace, do not end a chunk.) Example:

```
prepare Result := 0;
    Done := False;
    if not Initialized then
        Initialize;
    end if;

test ...
```

The chunk of code which should become the ‘prepare part’ of a test case begins with ‘Result :=...’ and ends with ‘end if;’ in the sixth line, then comes the next section, the ‘test part’. (More on the meaning of these ‘parts’ later.)

tg is not case-sensitive, just like the Ada language itself. You may write the tg keywords in upper, lower, or mixed case, just as your taste and coding conventions suggest. Comments take the usual Ada form (-- ...) and should begin in column 1. Comments which do not begin in column 1 can be copied into the test driver source code and cause confusion there (see Section 6.1 [Known Bugs], page 14).

2.2 The Global Section

The possible subparts of the global section are fail_handling, error_handling, context, exceptions, and define. They are explained subsequently, in that order. (Note that also in the script file the order of these parts may not be different. This might change in future versions of tg.) Apart from the context subpart, all of these are optional.

Remember also that all the keywords which start a subpart must begin in column 1.

• fail_handling ( stop | continue )

Specify what the driver should do if a test case fails. The alternatives are to stop execution after the first failing test case, or to continue execution regardless of the test case results. Default is continue.
• error_handling (stop | continue)
  Similar to fail_handling above. Specify what to do if a test case results in an error. Default is stop.

• context clauses
  clauses is an arbitrary number of with-/use-clauses which will be used as context clauses for the driver. clauses may span several lines. If it does, all but the first line must be indented.
  There must be a context clause here which makes visible subprograms Put_Line and New_Line, since the test driver performs its output through these. They will generally be those defined in Ada.Text_IO, thus you should normally write
  
  context with Ada.Text_IO; use Ada.Text_IO;

  in the context subpart. But you may also provide alternate versions for Put_Line and New_Line.

• exceptions list
  list must be a comma-separated, semicolon-terminated list of Ada exception identifiers. These exceptions will be ‘monitored’ by the driver, which means that a specific handler for each of these will be generated in each place where exceptions are checked for.
  This feature was necessary in the old, Ada 83 version of tg to find the names of exceptions that were raised. It is no longer needed in the current, Ada 95 version. tg creates others-handlers wherever they are necessary, and uses the predefined package Ada.Exceptions to get the exception names.

• define lines
  Subpart for global definitions. lines will be placed into the declarative part of the generated driver. You will typically define objects and subprograms here which are needed by the test cases. Each of the test cases can also have its own define part for data or subprograms needed only by that individual test case. Example:

  define Exit_Status : Integer; -- used by all the test cases

  function Result_Is_Correct (R : Result_Type) : Boolean is
  begin
    ...
  end Result_Is_Correct;

2.3 The Test Case Section

   The test case section of the script begins after the last part of the global section. It may contain an arbitrary number of test case descriptions and code parts.

2.3.1 Test Case Descriptions

   A test case description represents a single test case. A tg test case is characterized as follows:
   a. It forms an Ada block of its own, thus it is possible to declare any data objects or subprograms needed for that particular test case. You can do this in the define part (which resembles the define part of the global section).
   b. Before the actual test call, you might want to make some preparations. You can specify a prepare part, which is a chunk of code that the driver executes before the test call.
   c. The actual test part consists of a single Ada statement, typically a subprogram call to the software item under test. Hence, another word for the test part is test call.
d. After the test call has been executed, the driver checks whether it produced the expected result. There are two elements of that result:

1. The program path that was taken after the test call. Execution might have continued normally, or the test call might have raised an exception. The test driver automatically determines the taken path and stores it in the form of a string for future use. You don’t need to create any exception handlers yourself.

2. The second element of the result is the value of an arbitrary predicate. You may specify any boolean Ada expression which the driver checks after the test call. It may be a simple check for the value of a variable, or a call to a complex function of type Boolean.

e. The driver reports the result of the test case (pass/fail) to the standard output stream. You may specify the verboseness of that report. It is, for example, possible to suppress the report of passing test cases completely, while getting a full description of what happened in the event of a failing test case. You set the verboseness through command line options at translation time (see Chapter 3 [The tg Command], page 8).

f. Finally, you might have to clean up things. You might, for example, want to delete any files created during the test, etc. You can do this in the optional cleanup part of each test case.

The pattern explained above is precisely reflected in the format of test case descriptions. It looks like this:

```
***** test-case-title
define definitions
prepare preparations
test test-statement
pass path | predicate | path, predicate
cleanup cleanup-code
```

The meaning of the subparts define, prepare, and cleanup is clear from what has been said so far. All of them are optional. The title line, the test part and the pass-clauses, which are mandatory for every test case, are explained below.

For complete example test cases, See Chapter 5 [A Complete Example], page 12.

2.3.1.1 Test Case Titles

The start of a test case description is marked by the “keyword” *****, which also serves as an optical marker in the script file. The rest of the line is the test case title. It should explain briefly what is tested in that test case, allowing to decide quickly where the error lies, should this test case fail. Example:

```
***** function List_Length: List of length zero
```

tg numbers the test cases from 1 to n. It is recommended that you insert the number of the test case into the title as well, because this makes it easier to find a particular test case later. A parenthesized number at the beginning of the title is recognized by tg as the test case number. Example:

```
***** (17) function List_Length: List of length zero
```

It is, of course, easy for these numbers to become inconsistent during the writing of a script file. Therefore, there is a special Test Script Mode for Emacs, which provides a command to insert and correct the numbers automatically. If everything is installed correctly, and if you use the suffix .ts for your script files, you only need to type C-c C-n in Emacs to get the correct numbers.
2.3.1.2 The Test Call

tg allows an arbitrary chunk of Ada code in the test part. But in general it should only contain a single statement, because this makes it much easier to find out what happened during the test (especially if the test call raises an exception).

The results of the test call should be stored in variables local to that test case, such that they can later be checked in the pass-clause(s).

Example:

```
test Result := Test_Item (Some_Parameter);
```

2.3.1.3 Pass Clauses

There may be an arbitrary number of pass clauses after the test part. The result of the test case is “pass” if any one of these clauses is met. A pass-clause may have any of the following three forms:

```
pass path
pass predicate
pass path, predicate
```

`path` indicates which path the program should take after the test call. Its value may be either `='; which stands for the normal path (no exception was raised), or exception `exception-name`, to indicate that the test call should raise the named exception. `exception-name` must be an exception identifier which is visible in the test case description. You may use expanded (dot) notation here. If there is no `path` indication, it defaults to `='.

`predicate` must be an Ada expression yielding a boolean value. It may span several lines (continuation lines indented, as always), and tg uses it as the condition of an if statement. If no predicate is given, it defaults (effectively) to True.

A pass clause is said to be met if both the actual program path is correct and the predicate yields True.

Examples:

```
pass Number_Of_Elements = 5
pass exception Constraint_Error
pass exception IO_Exceptions.Name_Error, Analyze_Result
pass Status = True
    and then Is_Empty (List)
pass =>, Max = 10.23 -- '=>' is not required here
pass => -- the simplest pass-clause
```

2.3.2 Code Parts

You may insert code parts between the test cases to do additional work. For example, you might want to initialize a package before doing the actual testing. The syntax is fairly simple:

```
code lines
```

tg wraps Ada blocks around code parts and adds an exception handler at the end of each. Should an exception propagate out of a code part, the driver reports it as an “error”. The
global parameter error_handling (also see Section 2.2 [The Global Section], page 3) determines whether execution of the driver continues after this event or not.

Example:

code Init;
   Put_Line ("Package initialized.");

   if Tasking_Status /= Running then
      Put_Line ("Tasking is off.");
   end if;

   Put_Line ("Now continuing/starting with the test cases.");
3 The tg Command

The syntax for the tg command is

```
tg [options] script_file [driver_file]
```

In its simplest form, tg only takes one argument, the name of the script file. It reads and translates this script, writing the output to a file with the same name, but the suffix of the script (‘.ts’) changed to ‘.adb’. Example:

```
tg demo.ts
```
yields ‘demo.adb’.

You may provide an explicit output file name as the last argument:

```
tg demo.ts driver.adb
```

The options set the verboseness of the driver output.

- **p setting**
  Determines how the driver reports passing test cases.
  *setting* may be one of

  - *off*: No output.
  - *numbers*: Only test case numbers, followed by the string “pass.”.
    This is the default.
  - *titles*: Numbers and titles, followed by “...pass.” on the next line.
  - *full*: Numbers and titles, “...pass.” on the next line, and a short explanation on the line below that.

- **f setting**
  Same as -p, but for failing test cases.
  Default is full.

Examples:

```
tg -p full -f full demo.ts
```

```
tg -p off demo.ts
```
4 Drivers

The output of \texttt{tg} is the Ada source code of a single main program, the test driver. You compile it, link it to the test item, and execute the resulting program in order to perform the test.

You can have a look at the source code of the driver if you wish, to see how \texttt{tg} assembled your various code pieces, but there is no need for you to deal with this source code by any means. It is not intended to be human-readable. If you need to change something, you should modify the test script from which \texttt{tg} generated the driver.

There are, however, a few internal functions and workings, of which you might want to make use in your test scripts. These are described below.

4.1 Structure

A test driver generated by \texttt{tg} has the following structure:

\begin{verbatim}
-- header comment

with ...; use ...; -- from the context clause

procedure <name_of_script> is

  package Driver_Internals is
    -- ...
  end Driver_Internals;

  -- ...

-- global define part here

  -- ...

  package body Driver_Internals is
    -- ...
  end Driver_Internals;

begin

  -- ...

  -- code of the test cases here

  -- ...

exception

  -- default exception handlers

end <name_of_script>;
\end{verbatim}

The nested package \texttt{Driver_Internals} contains various status variables and routines to access them. They are listed and explained in the next section.

4.2 Example

\texttt{tg} translates the test case description
***** X = 3

define Result : Positive;
test Result := Subject(3);
pass exception Another_Error

into the following code (slightly reformatted make it more readable):

-- Test Case (3) X = 3
declare
   Result : Positive;
beg -- test case
   begin -- test part
      Result := Subject(3);
      Driver_Internals.Set_Path ("=>");
   exception
      when Another_Error =>
         Driver_Internals.Set_Path ("Another_Error");
      when E: others =>
         Driver_Internals.Set_Path (Ada.Exceptions.Exception_Name (E));
   end; -- test part
   begin -- result part
      if Driver_Internals.Path_Was ("Another_Error") then
         Driver_Internals.Test_Case_Passed := True;
         Put_Line ("(3) pass.");
      else
         Driver_Internals.Test_Case_Passed := False;
         Driver_Internals.Fail_Result := True;
         Put_Line ("(3) X = 3");
         Put_Line ("...FAIL.");
         Put_Line ("( & " & "path '" & Driver_Internals.Taken_Path & "' when 'Another_Error' was expected" & ")");
      end if;
   exception
      when Driver_Internals.Program_Terminate =>
         raise;
      when E: others =>
         Driver_Internals.Unexpected_Error := True;
         Put_Line ("ERROR: exception " & Ada.Exceptions.Exception_Name (E) & " raised in result part of test case 3.");
   end; -- result part
end; -- test case

4.3 Status Information

The package Driver_Internals, which is contained in the declarative region of the test driver program, exports the following declarations:

- function Passed return Boolean;
- function Failed return Boolean;

These functions indicate the result of the current test case. You may use them in the cleanup part, or in a subsequent code section, to take different action depending on the result.
• function Taken_Path return String;
  The path that the program took after the last test call. The value is either “=>”, if no
  exception was raised, or the name of the exception.

• function Path_Was (Path : in String) return Boolean;
  Function for comparing the Taken_Path (see above) with a given value.

• Program_Terminate : exception;
  This exception is being propagated by all handlers, and caught at the top level to perform
  a graceful exit. You may raise it yourself if you wish.

• The other functions declared in package Driver_Internals are intended for use by the
  generated code only.

In the following example, the driver provides additional information if the test case fails.

```
***** X = 1
define Result : Positive;
prepare Result := 1;
test Result := Subject(1);
pas Result = 3
cleanup if Driver_Internals.Failed then
  Put_Line (**Actual Result: " &
            Integer'Image(Result));
end if;
Result := 1;
```
5 A Complete Example

We want to test a single Ada function Subject, contained in the package Under_Test.

```ada
package Under_Test is

  Strange_Error,
  Another_Error,
  Illegal_Parameter : exception;

  function Subject (X : in Positive) return Positive;

end Under_Test;
```

Subject is required to return one if X is one, and raise exceptions Strange_Error, Another_Error, or Illegal_Parameter if X is two, three, or greater than three, respectively.

The following script describes an appropriate test:

```ada
-- FILE: example.ts
context with Text_IO; use Text_IO;
with Under_Test; use Under_Test;

exceptions Strange_Error, Another_Error, Illegal_Parameter;

***** X = 1
define Result : Positive;
test Result := Subject(1);
pass Result = 1

***** X = 2
define Result : Positive;
test Result := Subject(2);
pass exception Strange_Error

***** X = 3
define Result : Positive;
test Result := Subject(3);
pass exception Another_Error

***** X = 4
define Result : Positive;
test Result := Subject(4);
pass exception Illegal_Parameter

***** X = Positive'Last
define Result : Positive;
test Result := Subject(Positive'Last);
pass exception Illegal_Parameter
```

You can translate example.ts by issuing the command

tg example.ts

This produces example.adb, the source code of the driver. You have to compile it and link it with package Under_Test. Executing the resulting program then produces the following output.
Chapter 5: A Complete Example

(1) pass.
(2) pass.
(3) pass.
(4) pass.
(5) pass.

Total test result: pass.

Now suppose that in test case (3), Subject raised exception Illegal_Parameter by mistake. Then the output would be

(1) pass.
(2) pass.
(3) X = 3
   ...FAIL.
   (path 'Illegal_Parameter' when 'Another_Error' was expected)
(4) pass.
(5) pass.

Total test result: FAIL.
6 Known Bugs and Restrictions

There are some minor issues with the current version of 
tg you should be aware of.

6.1 Known Bugs

Ada Comments (marked by --) which do not begin at column 1 may cause 
tg to generate incorrect Ada code. Consider the following example.

***** X = 1
define Result : Positive;
test Result := Subject(1);
pass Result = 1
   -- if Result is not 1, something is badly broken!

While 
tg runs without complaining, the Ada compiler reports a syntax error: a missing then.

If you really need test-case specific comments in the driver generated by 
tg (e.g., because you distribute the Ada sources without 
tg or the *.ts test scripts), write comments after define or test, but not after pass. The following example generates a syntactically correct test driver.

***** X = 1
define Result : Positive;
   -- some test-case specific comments
test Result := Subject(1);
   -- other test-case specific comments
pass Result = 1

6.2 Restrictions

• 
tg mercilessly overwrites any file with the same name as the test 
driver to be generated. Thus, if you have written a package some_unit with the body in some_unit.adb and a test script some_unit.ts, then running 
tg some_unit.ts overwrites some_unit.adb. Your source code has been lost! It is recommended to always use a unique name for the test script, e.g. some_unit_ut.ts for unit-tests.

• 
tg does not check or prevent the hiding of identifiers. E.g. if your test script declares

define Driver_Internals : Positive;

then 
tg generates a test_driver, but the Ada compiler will report syntax errors. (In practice, don’t worry. Such collisions hardly ever happen without malice.)

• 
tg does not warn about inconsistent test case numbers, as, e.g., below.

   -- first test case
   ***** (3)
   [...]

   -- second test case
   ***** (3)
   [...]

Inconsistent test case numbers in the test script have no consequences for the function of the test driver. But when the driver reports a test case to pass or fail, it uses consistent numbers. This may be confusing for the user.

• 
tg does not try too hard to provide meaningful error messages on incorrect inputs. E.g., if you write 
tg -f number x.ts on the command line, then 
tg raises a constraint error. (The correct input would be 
tg -f numbers x.ts.)
<table>
<thead>
<tr>
<th>Concept Index</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong></td>
</tr>
<tr>
<td>Ada</td>
</tr>
<tr>
<td><strong>B</strong></td>
</tr>
<tr>
<td>bugs (known)</td>
</tr>
<tr>
<td><strong>C</strong></td>
</tr>
<tr>
<td>case-sensitivity</td>
</tr>
<tr>
<td>cleanup part</td>
</tr>
<tr>
<td>code part</td>
</tr>
<tr>
<td>comments in test scripts</td>
</tr>
<tr>
<td>context clause</td>
</tr>
<tr>
<td><strong>D</strong></td>
</tr>
<tr>
<td>declarations (global)</td>
</tr>
<tr>
<td>declarative part (global)</td>
</tr>
<tr>
<td>define part (test case)</td>
</tr>
<tr>
<td><strong>E</strong></td>
</tr>
<tr>
<td>error (result of a test case)</td>
</tr>
<tr>
<td>exception (raised by the test call)</td>
</tr>
<tr>
<td>exception (raised in a code part)</td>
</tr>
<tr>
<td>execute a test handlers</td>
</tr>
<tr>
<td>execute a test case</td>
</tr>
<tr>
<td>execute a test driver</td>
</tr>
<tr>
<td><strong>F</strong></td>
</tr>
<tr>
<td>fail (result of a test case)</td>
</tr>
<tr>
<td>format of test scripts</td>
</tr>
<tr>
<td><strong>G</strong></td>
</tr>
<tr>
<td>global declarations</td>
</tr>
<tr>
<td>global section</td>
</tr>
<tr>
<td>global section (test script)</td>
</tr>
<tr>
<td><strong>K</strong></td>
</tr>
<tr>
<td>keywords</td>
</tr>
<tr>
<td><strong>N</strong></td>
</tr>
<tr>
<td>Native language of tg</td>
</tr>
<tr>
<td>number (of a test case)</td>
</tr>
<tr>
<td><strong>O</strong></td>
</tr>
<tr>
<td>output format</td>
</tr>
<tr>
<td><strong>P</strong></td>
</tr>
<tr>
<td>pass (result of a test case)</td>
</tr>
<tr>
<td>pass clause</td>
</tr>
<tr>
<td>path</td>
</tr>
<tr>
<td>path (taken in a test case)</td>
</tr>
<tr>
<td>perform a test</td>
</tr>
<tr>
<td>perform a test case</td>
</tr>
<tr>
<td>predicate (in a pass clause)</td>
</tr>
<tr>
<td>predicate (result part)</td>
</tr>
<tr>
<td>prepare part</td>
</tr>
<tr>
<td><strong>R</strong></td>
</tr>
<tr>
<td>result of a test</td>
</tr>
<tr>
<td>result of a test call</td>
</tr>
<tr>
<td>result of a test case</td>
</tr>
<tr>
<td>result part</td>
</tr>
<tr>
<td>results</td>
</tr>
<tr>
<td><strong>S</strong></td>
</tr>
<tr>
<td>syntax (of a test case description)</td>
</tr>
<tr>
<td>syntax (of the tg command line)</td>
</tr>
<tr>
<td><strong>T</strong></td>
</tr>
<tr>
<td>test</td>
</tr>
<tr>
<td>test call</td>
</tr>
<tr>
<td>test case</td>
</tr>
<tr>
<td>test case description</td>
</tr>
<tr>
<td>test case number</td>
</tr>
<tr>
<td>test case result</td>
</tr>
<tr>
<td>test case section</td>
</tr>
<tr>
<td>test case section (test script)</td>
</tr>
<tr>
<td>test case title</td>
</tr>
<tr>
<td>test driver</td>
</tr>
<tr>
<td>test item</td>
</tr>
<tr>
<td>test part</td>
</tr>
<tr>
<td>test results</td>
</tr>
<tr>
<td>test script</td>
</tr>
<tr>
<td>test script mode (Emacs)</td>
</tr>
<tr>
<td>tg (command line syntax)</td>
</tr>
<tr>
<td>title (of a test case)</td>
</tr>
<tr>
<td>ts-mode</td>
</tr>
</tbody>
</table>
Keyword Index

*                                5
***** ..................................... 5

C
cleanup .................................... 5
code ......................................... 6
context ....................................... 4

D
define (global) ......................... 4
define (test case) .................... 5

E

error_handling ......................... 3, 6
exceptions .............................. 4

F
fail_handling ............................ 3

P
pass .......................................... 6
prepare .................................... 5

T
test .......................................... 5
# Table of Contents

1 Introduction ......................................................... 1  
   1.1 A Small Example................................................. 1  
   1.2 Testing Terminology............................................ 1  

2 Test Scripts .......................................................... 3  
   2.1 Test Script Basics .............................................. 3  
   2.2 The Global Section ............................................. 3  
   2.3 The Test Case Section ......................................... 4  
      2.3.1 Test Case Descriptions .................................. 4  
      2.3.1.1 Test Case Titles ...................................... 5  
      2.3.1.2 The Test Call .......................................... 5  
      2.3.1.3 Pass Clauses ........................................... 6  
   2.3.2 Code Parts ................................................... 6  

3 The tg Command ....................................................... 8  

4 Drivers ............................................................... 9  
   4.1 Structure ....................................................... 9  
   4.2 Example ........................................................ 9  
   4.3 Status Information ............................................. 10  

5 A Complete Example ................................................ 12  

6 Known Bugs and Restrictions ....................................... 14  
   6.1 Known Bugs ..................................................... 14  
   6.2 Restrictions ................................................... 14  

Concept Index ......................................................... 15  

Keyword Index ......................................................... 16