tg

A Simple Test Driver Generator for Modula-2 programs
(Version 3.1)

André Spiegel, Ralf Reißing
University of Stuttgart, Germany

1 Introduction

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

If you want to test something, for example an Modula-2 module, 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 procedure call to some routine of the module, 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 Modula-2, but you might also use it to test software written in other programming languages. (This would require 'interfacing' that software to Modula-2, since the test driver is always a Modula-2 program.) `tg` itself is written in Ada.

1.1 A Small Example

Suppose you want to test a Modula-2 function which counts the number of '`&`' characters within a given string.

```
PROCEDURE CountAmpersand (Str: ARRAY OF CHAR): CARDINAL;
```

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 like the following:

```
WriteString("Testing three ampersands in the middle... ");
WriteLn;
Count := CountAmpersand("abc&&&abc");
IF (Count = 3) THEN
  WriteString("pass."); WriteLn;
ELSE
  WriteString("fail."); WriteLn;
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 := CountAmpersand("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 procedure, or a module, 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 procedure 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 `VAR` 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 driver determines 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*.
- 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.

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 Modula-2 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 Modula-2 code, marked by keywords which tell `tg` at where it should insert that code into the driver. 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;
test ...
```

The chunk of code which should become the 'prepare part' of a test case begins with '`Result :=...`' and ends with '`END;`' in the sixth line, after this follows the next section, the 'test part'. (More on the meaning of these 'parts' later.)

`tg` is not case-sensitive. You may write the `tg` keywords in upper, lower, or mixed case, just as your taste and coding conventions suggest. Comments take the usual Modula-2 form (`(* ... *)`), but may not be nested. Note that comments which apply to the *script* should begin in column 1, otherwise `tg` might consider them part of a code chunk and copy them into the driver program (which could be confusing if someone ever examines the driver code).

2.2 The Global Section

The possible subparts of the global section are `fail_handling`, `context`, 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`.

- `context` *clauses*

  clauses is an arbitrary number of `IMPORT`-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. Type definitions and definitions of constants should be put here, too. Also procedures needed for your test cases can be defined here.

```
context IMPORT AModule;
        TYPE   AType = ...;
        CONST  AConstant = ...;
        PROCEDURE ResultIsCorrect (R: ResultType): BOOLEAN;
        BEGIN
          ...
        END ResultIsCorrect;
```

- `define` *lines*

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

```
define ExitStatus: INTEGER;  (* used by all the test cases *)
```

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 a Modula-2 procedure of its own, thus it is possible to declare any local variables 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 (typically) of a single Modula-2 statement, typically a procedure 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. The result is the value of an arbitrary predicate. You may specify any boolean Modula-2 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 7).

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`	*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 11.

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:

```
***** PROCEDURE ListLength: List of length zero
```

2.3.1.2 The Test Call

`tg` allows an arbitrary chunk of Modula-2 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.

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 := TestItem(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 *all* of these clauses are met. A pass-clause has the following form:

`pass` *predicate*

predicate must be a Modula-2 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.

A pass clause is said to be met if the predicate yields `TRUE`.

Examples:

```
pass NumberOfElements = 5
pass Status = True
     AND IsEmpty(List)
```

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 module before doing the actual testing. The syntax is fairly simple:

`code` *lines*

Example:

```
code Init;
     WriteString("Module initialized."); WriteLn;
     WriteString("Now starting with the test cases."); WriteLn;
```

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 ‘`.mod`’. Example:

```
tg demo.ts
```

yields ‘`demo.mod`’.

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

```
tg demo.ts driver.mod
```

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 `tg` is the Modula-2 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 `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 `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 `tg` has the following structure:

```
(* header comment *)

MODULE <name_of_script> is

(* code from the context clause here *)

(* global define part here *)

(* test cases and code parts as procedures *)

BEGIN
  (* calls to test cases and code sections *)
END <name_of_script>.
```

4.2 Status Information

The test driver module contains two predefined global variables:

`Fail_Result: BOOLEAN`
is TRUE if at least one of the previous test cases failed, else FALSE

`Test_Case_Passed: BOOLEAN`
is TRUE if the last test case was passed, else FALSE. You may want to take different actions in the cleaup part of a test case dependent on its result.

5 A Complete Example

We want to test a single Modula-2 function `Subject`, contained in the module `UnderTest`.

```
DEFINITION MODULE UnderTest;

  PROCEDURE Subject (X : in CARDINAL): CARDINAL;

END UnderTest;
```

`Subject` is required to return 1 if `X` is a multiple of 2, else `X` itself.

The following script describes an appropriate test:

```
(* FILE: example.ts
  ...
*)
context IMPORT Under_Test;

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

***** X = 2
define Result: CARDINAL;
test   Result := Subject(2);
pass   Result = 1

***** X = 3
define Result: CARDINAL;
test   Result := Subject(3);
pass   Result = 3

***** X = 16
define Result: CARDINAL;
test   Result := Subject(16);
pass   Result = 1

***** X = MAX(CARDINAL)
define Result: CARDINAL;
test   Result := Subject(MAX(CARDINAL));
pass   Result = 1
```

You can translate `example.ts` by issuing the command

```
tg example.ts
```

This produces `example.mod`, the source code of the driver. You have to compile it and link it with module `UnderTest`. Executing the resulting program then produces the following output

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

Total test result: pass.
```

Now suppose that in test case (3), `Subject` returned 1 instead of 3. Then the output would be

```
(1) pass.
(2) pass.
(3) X = 3
     ...FAIL.
        (predicate is FALSE)
(4) pass.
(5) pass.

Total test result: FAIL.
```

Table of Contents