— DejaGnu —
a short introduction

Randolf Rotta

July 26, 2002

The single most important rule of testing is to do it.
— Brian Kernighan and Rob Pike, The Practice of Programming —

Abstract

This article tries to highlight the importance of regression testing and will explore the functionality of the regression test framework DejaGnu. It's intended as a short tutorial on how to use DejaGnu.

1 Regression tests

1.1 What's that?

The purpose of regression tests is mainly to ensure that a program has not regressed. This means, the functionality and behavior of a software system or a specific part of a system hasn't changed or at least doesn't regress while changes where done in the source code. There are two common types of tests often called regression test.

Functional tests check the entire system to meet external requirements and goals like performance. They were specified by the customer and permit him to understand and track a project's growth. Here real world data can be used as input for the system and then the output can be compared with the expected results. Sometimes there are special specifications on the functionality, which also provide good test cases.

Another source of test cases for functional tests are interface or language specifications which have to be implemented. For example for the GNU Debugger (gdb) it is essential, that it's interface works as specified in the documentation because other tools like graphical debuggers and IDEs use gdb as debugging backend.

Unit tests were created by the programmers to check every aspect of a single part of the system like a class or module on expected behavior. The goal is to test each part (unit) of code in isolation. This is very important, because it directly supports the programmer in his everyday work.

1.2 Why should I test?

The results of test runs provide a documented stage of development by showing which parts of the system work and which don't work yet. So tests supply the programmer a documented level of correctness. Moreover the test cases itself provide another form of documentation as everybody can examine the test to learn more about what a unit should do and how it is used.

In conjunction with versioning control systems like CVS regular tests can indicate when and where new errors were introduced. They are likely to be caused by the code changed since the last check point.

This allows save refactoring of the source code from little code clean ups up to the replacement of complete parts of a system without unnoticed changes in behavior and functionality. Now the programmer has the ability and freedom to change and reorganize even complex systems in the most straight way without worrying how to find back to the old behavior and functionality. By the way, this is the main reason why regression tests are a so essential part of the eXtreme Programming methodology.

An often used approach is to write the test cases of a unit before the actual coding. So one has to think about, what the unit should do and which problems could arise, very early, which can lead to a better design in the first place. After writing the test cases, they should be tested by running them without the real unit to see if they actually fail, otherwise they may be wrong. Then one can work on the unit until all test cases succeed. So the programmer know, when the necessary work is done. Theoretically the resulting code can be seen bug free, since the code reacts like it was desired and expressed in the test cases.

In order to be able to perform the test cases, the programs need an interface which permits automated test in some way, thus leading to a *design for testing*. The general interfaces of the programs can also benefit from this.

1.3 So for what testing frameworks?

Doing regression tests means running many different test cases and running them often. So it's unacceptable doing them manually, because this would be very time consuming and also unreliable. What is needed, are automated exercises of the code.

In some programming languages it seems easy to write test directly in the same language. For example JUnit helps doing regression test in Java, but different languages like C have exceptions which cannot be caught from within the program. Reacting on segmentation faults, race conditions and so on or performing network wide or cross platform tests needs a testing framework.

Also most times tests are performed by giving some input to the application and compare the resulting output against expected results. For textual output, comparison can be done with regular expressions to only match the interesting parts of the data and not the syntactical sugar around it.

There exists quite a big number of testing frameworks and supporting programs. Naming them all isn't possible here, so only some interesting open source products will be mentioned. A summary of commercial tools can be found on http://www.aptest.com/ and http://www.testingfaqs.org/.

DejaGnu	general purpose test framework for batch and interactive programs
STAF	Software Testing Automation Framework
Check	very easy framework for defining unit tests
Android	GUI testing tool based on Tcl and Expect
JUnit	a testing framework for Java
Abbot	Java GUI testing
Siege	HTTP-Based testing of servers and web sites
LogiTest	regression testing of web sites with GET, POST, Cookies, HTTPS etc.

2 Interfaces for testing

2.1 Batch oriented tests

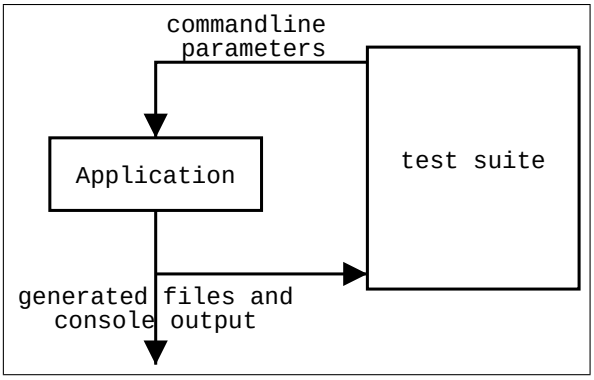

Figure 2.1: Batched testing

When the only interface to an program is it's command line, then the only thing automated test suites can do, is to start the program with given parameters and examine the output and maybe generated files. For example compilers are such programs where no interaction is required.

Batch oriented test are easy to write and easy to be done automatic, but it's difficult to test single units of a large system this way, when there is no suitable way of accessing them over the command line parameters.

2.2 Stream based tests

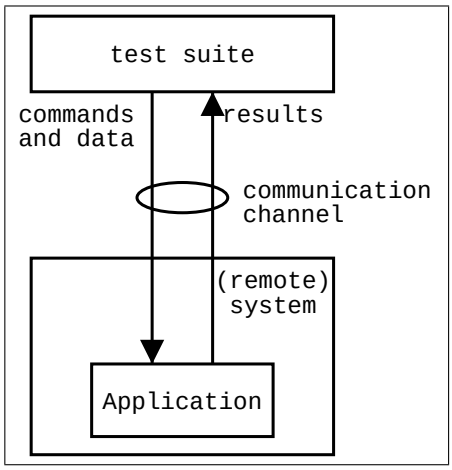

Figure 2.2: Testing over streams

This covers quite a big range of test methods. The first coming in mind are console based programs which interact with the user over its standard input and output channels AKA STDIN and STDOUT. Most of this programs take a line of input as command then do something and put out some messages with the results. Examples for such programs are mathematical engines like singular, octave or scilab, or debuggers like the GNU Debugger GDB. As mentioned earlier it may be required that interfaces react as specified through a language definition or similar things.

But this method isn't restricted to direct text based channels. Also web-servers and web sites can be tested by using HTTP GET and POST requests for communication.

2.3 GUI testing

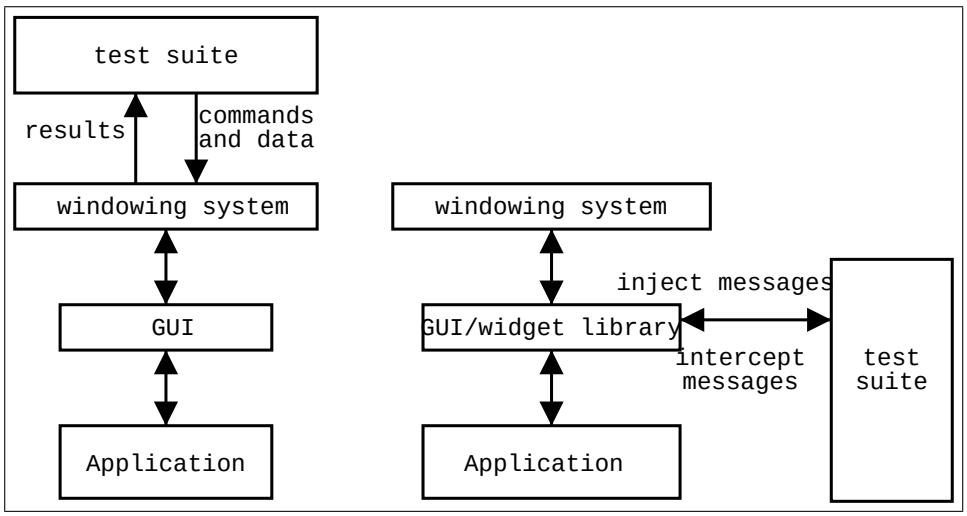

Figure 2.3: capture/replay based vs. indirect tests

Testing graphical user interfaces has many problems. Although there exists tools for testing GUIs by replaying previously captured actions like keystrokes and mouse events and comparing screenshots of windows, they can't cope well with even lightly changed arrangement of the GUI elements. So the test cases have to be captured again after every change of the user interface. Stream based tests work around this problem by using regular expressions for matching the results.

A better way for automated testing of graphical user interfaces is to generate events in the application itself. So for example buttons can be "clicked" by sending a message to the button and data for checking the results of a test case can be obtained by querying the properties of widgets. Thereby the test become independent to the arrangement of the widgets on the screen. Of course this needs the support of the used widget library. In contrast capture/replay based tools often only need the XTEST extension of the used X-Server to emit X-Events for communicating with GUIs.

2.4 Embedded test code and interfaces

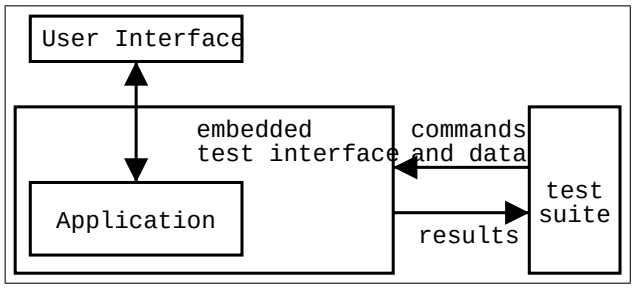

Figure 2.4: embedded test interfaces

In the case where simple batch or stream based tests over the user interface of the application aren't possible because the interface doesn't provide enough access or when a GUI should be tested, but the used widget set doesn't provide an applicable test interface, embedded additional testing interfaces in the application are very helpful.

It's a good idea, to split the tests in an stream interface embedded in the application and a normal extern test suite like DejaGnu, which provides the test data and checks the results. So test of the user interface and unit test can be incorporated in one single testing framework. Since Tcl is an embeddable scripting language, it could easily be used as a generic interface to the application supporting normal usage and testing.

3 DejaGnu

3.1 Introduction

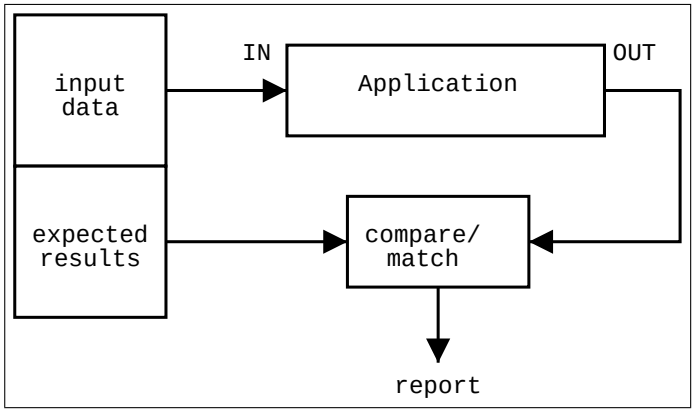

Figure 3.1: principle of DejaGnu

DejaGnu is a collection of functions for Tcl and Expect for testing other programs and tools. Each program is tested by one or more test suites, which do the tests expressed as expect-scripts. Most times this tests can be done by sending the application a command and data and matching the result against the expected result. For the matching of the results the advanced regular expressions of Tcl can be used.

DejaGnu is based on Expect, which itself is written in Tcl. All interaction with the application is done over Expect procedures, which use generic communication channels. So it's even possible to test applications on remote machines.

3.2 Features

Since DejaGnu is based on Expect, it can be used to test batch oriented and also interactive programs, as long as they provide a stream based interface. This are mainly console based programs and tools but also libraries can be tested by using little wrappers as mentioned in 2.4. Here some features:

- easy to write tests for batch oriented and for *interactive* programs
- provides layer of abstraction, necessary for *cross-platform* testing
- modular communication setup for *remote* testing
- unified, machine parse-able but also human readable output format
- conforms to POSIX 1003.3 standard for regression test frameworks

3.3 History

DejaGnu was mainly developed by Cygwin Support and used on BSD, but later ported to Linux and other operating systems. At present development is in progress for more support of realtime operating system and to integrate ExpectTk for capture/replay based GUI testing. By now DejaGnu is actively used in development of GDB (GNU Debugger), GCC-3.x (GNU Compiler Collection) and Cygwin (UNIX-like environment for Win32).

In 1987 Tcl (Tool command language) was developed by John Ousterhout as an embeddable command language for IC design tools in Berkeley. In this time graphical user interfaces weren't so widespread yet and they had several tools there, each with an own weak command language. The development of Tcl had following three goals in mind (cited from Tcl-Homepage):

- The language must be extensible: it must be very easy for each application to add its own features to the basic features of the language, and the application-specific features should appear natural, as if they had been designed into the language from the start.
- The language must be very simple and generic, so that it can work easily with many different applications and so that it doesn't restrict the features that applications can provide.
- Since most of the interesting functionality will come from the application, the primary purpose of the language is to integrate or "glue together" the extensions. Thus the language must have good facilities for integration.

By now Tcl is a very popular and widespread language, used for cross platform scripting and embedded command interfaces. Many people are also attracted from the Tk-extension, which provides an easy way to create graphical user interfaces.

Expect was first written as a tool for simple telnet automation by Scott Paisley and Don Libes. Later Don Libes was very impressed by Tcl's capabilities and generalized and extended the program forming expect as known now. It's now useful for automating interactive applications such as telnet, ftp, passwd, fsck, rlogin, tip, etc.

3.4 Supplied procedures

DejaGnu inherits a whole bunch of procedures from TCL and Expect and also provides a wide range of new functions for all forms of cross platform and netwide testing. As there are too many of them, only a few of the more used will be named here as examples. The users manual tries to document all of them and is a better reference than this tutorial.

`spawn`	starts a program
`pass`	declares a test to have passed
`fail`	declares a test to have failed
`note`	appends an informational message to the logfile
`send`	sends a string to the application
`expect`	analyzes the output of the application
`getdirs`	get a list of files and directories matching a pattern
`find`	file search
`diff`	finds differences of two or three files

There are some system and tool dependent procedures, which are used by DejaGnu if provided by the tester:

`tool_start`	starts the tool and initializes it; for a batch oriented tool it should be executed, leaving the output in the variable `comp_output`
`tool_load`	loads something for a particular test case into the tool
`tool_exit`	cleaning up before DejaGnu exits
`tool_version`	prints the version number for use in the summary report

4 An example: Mergesort

4.1 The program

For showing how to perform tests we naturally need something to be tested. As example the simple mergesort algorithm will be used. The algorithm has some appropriate properties making it a good example. It's very simple but consists of three distinct units which could be tested. Apart from that honestly merge sort shouldn't be used in real systems since there are faster sorting algorithms.

merge This function merges two sorted, adjacent ranges of the array into a single, sorted array. After that the region of the two ranges is overwritten by the sorted array. Since we will need temporary space here we provide it with a function parameter

split The splitting function gets a range and divides this into two halves, calls itself recursively for each half, if it contains more than one element and after that merges the two adjacent halves with the merge function.

mergesort This should be the end user interface to the sorting function. Here the temporary memory is allocated and then the splitting function called with the initial parameters.

So we can now define an interface and the internal structure of our unit as seen below. The variable `data` contains the table to be sorted as an array of `size` pointers to the real elements. The compare function will be used to compare two given pointers to elements. It's return value should be negative if `a` is less than `b`, zero if booth are equal and otherwise positive. The array should be sorted with lowest elements first by the help of the compare function.

File: libsort.h

```
#ifndef LIBSORT_H__
#define LIBSORT_H__

void mergesort(void** data, int size,
               int (*compare)(void *a, void *b));

/* internal function */
void merge(int lstart, int rstart, int stop, void **data, void **temp,
           int (*cmp)(void*,void*));
void split(int start, int stop, void **data, void **temp,
           int (*cmp)(void*,void*));

#endif
```

It would be hard to test the three functions in real isolation, but the calling dependencies suggest to test them in a particular order. `mergesort` calls `split` and `split` uses `merge`. So `merge` should be tested first, since all other functions are bound to fail if `merge` doesn't work correctly.

4.2 The test driver

After we specified the interface to the units and the test cases, we can now start coding a test driver, which we later use to communicate between the test suite and our real code. The program will read all parameters from `argv` into an array and then call the specified function with the right parameters. After that, the resulting array is printed out to be checked by the test suite.

File: sorttest.c

```
#include <stdio.h>
#include <string.h>
#include "libsort.h"

int compare(int *a, int *b)
{
        return *a - *b;
}

int main(int argc, char *argv[])
{
        int size = argc - 2;
        int **data = (int **)malloc(sizeof(int *) * size);
```

```
        void **temp = (void **)malloc(sizeof(void *) * size);
15      int **realdata;
        int i;

        /* reading the data */
        for (i=0; i<size; i++) {
20              data[i] = (int *)malloc(sizeof(int));
                *data[i] = atoi(argv[i + 2]);
        }

        /* run the test */
25      switch (atoi(argv[1]))
        {
        case 0: /* merge */
                realdata = data + 3; size -= 3;
                merge(*data[0], *data[1], *data[2],
30                    realdata, temp, compare);
                break;
        case 1: /* split */
                realdata = data + 2; size -= 2;
                split(*data[0], *data[1], realdata, temp, compare);
35              break;
        case 2: /* mergesort */
                realdata = data + 1; size -= 1;
                mergesort(realdata, *data[0], compare);
                break;
40      default:
                printf("unknown function!\n");
        }

        /* generate output */
45      for (i=0; i<size; i++) printf("%i ", *realdata[i]);
        printf("\n");

        return 0;
    }
```

4.3 Determining possible test cases and writing a test suite

Sometimes test cases can be generated from the functionality description of the unit to be written. But often a little bit black magic and guessing is needed to find test cases which probably uncover possible errors. There also exists a whole testing methodology with several approaches to test case generation, but this may be to much for this tutorial.

So we keep guessing and pick some special cases for our input data and find the expected results by using common sense. The following table shows some test cases.

unit	**parameters**	**data**	**expected result**

`merge(s1, s2, stop, data)`	`0 1 2`
	`0 1 2`
	`0 4 5`
	`0 1 5`
	`4 5 6`
		`1 2`
		`2 1`
		`1 2 3 4 0`
		`4 0 1 2 3`
		`4 3 2 1 6 5`
			`1 2`
			`1 2`
			`0 1 2 3 4`
			`0 1 2 3 4`
			`4 3 2 1 5 6`

`split(start, stop, data)`	`0 1`
	`0 2`
	`0 3`
	`3 6`
		`5`
		`2 -2`
		`3 2 1`
		`3 2 1 6 5 4`
			`5`
			`-2 2`
			`1 2 3`
			`3 2 1 4 5 6`

`mergesort(size, data)`	`4`
	`4`
	`4`
	`3`
	`3`
		`1 1 1 1`
		`1 2 3 4`
		`4 3 2 1`
		`-2 -5 -6`
		`-6 -5 -2`
			`1 1 1 1`
			`1 2 3 4`
			`1 2 3 4`
			`-6 -5 -2`
			`-6 -5 -2`

With the help of the test driver and the found test cases we can now write a test suite which later will check our sorting functions. For simplicity the test suite consists of two parts. There is an array which contains the test data. The first column names the test case, the second contains to command line parameters for *sorttest* and the last column is a regular expressions pattern which must match the output of *sorttest* to pass the test. The second part of the test suite is a loop which takes each test case from the array and executes it. But let's look at the code first. The file must be placed in the subdirectory testsuite/sorttest/ to work.

File: testsuite/sorttest/mergesort.exp

```
set testdata {
    {"merge1" "0  0 1 2  1 2" "^1 2.*$"}
    {"merge2" "0  0 1 2  2 1" "^1 2.*$"}
    {"merge3" "0  0 4 5  1 2 3 4 0" "^0 1 2 3 4.*$"}
    {"merge4" "0  0 1 5  4 0 1 2 3" "^0 1 2 3 4.*$"}
    {"merge5" "0  4 5 6  4 3 2 1 6 5" "^4 3 2 1 5 6.*$"}
    {"split1" "1  0 1  5" "5.*$"}
    {"split2" "1  0 2  2 -2" "^-2 2.*$"}
    {"split3" "1  0 3  3 2 1" "^1 2 3.*$"}
    {"split4" "1  3 6  3 2 1 6 5 4" "^3 2 1 4 5 6.*$"}
    {"sort1"  "2  4  1 1 1 1" "^1 1 1 1.*$"}
    {"sort2"  "2  4  1 2 3 4" "^1 2 3 4.*$"}
    {"sort3"  "2  4  4 3 2 1" "^1 2 3 4.*$"}
    {"sort4"  "2  3  -2 -5 -6" "^-6 -5 -2.*$"}
    {"sort5"  "2  3  -6 -5 -2" "^-6 -5 -2.*$"}
}
```

```
    global SORTTEST
    foreach pattern $testdata {
        eval "spawn $SORTTEST [lindex $pattern 1]"
        expect {
            -re [lindex $pattern 2] { pass [lindex $pattern 0] }
            default { fail [lindex $pattern 0] }
        }
    }
```

Note that *$testdata* is a two dimensional array. The lines contain the different test cases and the columns contain their data. The `eval` statement is used, because the parameters for the program to be started are created by the normal Tcl parameter splitting at white spaces. So `spawn` would pass all parameters retrieved by `lindex` as a single parameter. To avoid this in favor for an easier command line parsing in the test driver, firstly the `spawn` command is created as string and then split up and executed through `eval`. The used commands are shortly explained in the following table.

command	**meaning**
`foreach` *var array*	cycles through the body for each element of the *array*, containing it in *var*
`spawn` *program parameters*	spawn a new process of a *program* with the given *parameters*
`lindex` *array index*	returns the value of a *array* at *index*
`eval` *string*	evaluates a *string* as Tcl command
`expect`	analyzes the output of the last spawn process by the given rules

4.4 The sorting code

As the interface is defined now and the test cases ready, we should really start with the actual code. This is something one could easily do oneself, but following code will also just do right.

File: libsort.c

```
#include <stdlib.h>
#include <unistd.h>
#include "libsort.h"

void merge(int lstart, int rstart, int stop, void **data, void **temp,
           int (*cmp)(void*,void*))
{
        int lp = lstart;
        int rp = rstart;
        int pos;
```

```
        for (pos=0; lp<rstart && rp<stop;)
                if (cmp(data[lp], data[rp])<0)
                {
                        temp[pos++] = data[lp++];
                } else {
                        temp[pos++] = data[rp++];
                }
        while (lp < rstart)
                temp[pos++] = data[lp++];
        while (rp < stop)
                temp[pos++] = data[rp++];

        memcpy(data + lstart, temp, sizeof(void **) * pos);
}

void split(int start, int stop, void **data, void **temp,
                int (*cmp)(void*,void*))
{
        int middle = (stop - start) / 2;

        /* recursion step */
        if (middle-start>1)
                split(start, middle, data, temp, cmp);
        if (middle > start && stop-middle>1)
                split(middle, stop, data, temp, cmp);

        /* merge the two halfes */
        merge(start, middle, stop, data, temp, cmp);
}

void mergesort(void** data, int size,
               int (*compare)(void *a, void *b))
{
        /* setup of configuration */
        void **temp = (void **)malloc( sizeof(void *) * size);

        /* start mergesorting */
        split(0, size, data, temp, compare);

        /* clear configuration */
        free(temp);
}
```

4.5 Setting up configure and the makefile

Although it's also possible running a test suite with *runtest*, it's more easier to start it via a makefile. Since GNU Automake directly supports DejaGnu by the "check" target in Makefiles, we will activate this feature by adding the keyword "dejagnu" to the AUTOMAKE_OPTIONS variable. So in order to run the test suite only *make check* needs to be executed.

In order to use the powerful autoconf/automake scheme, we have to create a configure.in file which may look like following one:

File: configure.in

```
1   AC_PREREQ(2.5)
    AC_INIT(libsort.c)
    AM_CONFIG_HEADER(config.h)
    AM_INIT_AUTOMAKE(libsort,0.0.1)
5   AC_PROG_CC
    AC_OUTPUT(Makefile)
```

Then we need the Makefile.am file:

File: Makefile.am

```
1   AUTOMAKE_OPTIONS= dejagnu
    bin_PROGRAMS    = sorttest
    sorttest_SOURCES = sorttest.c libsort.c
    INCLUDES        =
5   LDADD           =
    CLEANFILES      = *~
    DISTCLEANFILES  = .deps/*.P
    EXTRA_DIST= testsuite

10  RUNTESTDEFAULTFLAGS = --tool sorttest SORTTEST=`pwd`/sorttest \
                          --srcdir $$srcdir/testsuite
```

4.6 Time for checking

Now, as we just wrote the last piece of code, we can try all out. Following commands will generate the configure script and other files:

```
aclocal
autoheader
autoconf
automake --add-missing --include-deps --foreign
```

In order to compile the program and run the tests, following commands are necessary:

```
./configure
make
make check
```

The last command ***make check*** started DejaGnu to run the test suite. A summary of the test results is shown and if all went right, one should see something similar to the following text, which also can be found in DejaGnu's summary file sorttest.sum.

```
...
Running probe/testsuite/sorttest/mergesort.exp ...
FAIL: split3
FAIL: split4
FAIL: sort2
FAIL: sort3
FAIL: sort4

                === sorttest Summary ===

# of expected passes            9
# of unexpected failures        5
make[1]: *** [check-DEJAGNU] Error 1
make[1]: Leaving directory ‘probe’
make: *** [check-am] Error 2
```

As probably everyone could see, the test suite discovered some errors in our code, which is also reflected by make's messages and return value. To see the test input and corresponding outputs, the sorttest.log file contains a complete log of the test run.

File: sorttest.log

```
1   Test Run By randolf on Thu Jul 25 17:42:24 2002
    Native configuration is i686-pc-linux-gnu

                    === sorttest tests ===
5
    Schedule of variations:
        unix

    Running target unix
10  Using /usr/share/dejagnu/baseboards/unix.exp
      as board description file for target.
    Using /usr/share/dejagnu/config/unix.exp
      as generic interface file for target.
    ERROR: Couldn't find tool config file for unix.
15  Running probe/testsuite/sorttest/mergesort.exp ...
```

```
    1 2
    PASS: merge1
    1 2
    PASS: merge2
20  0 1 2 3 4
    PASS: merge3
    0 1 2 3 4
    PASS: merge4
    4 3 2 1 5 6
25  PASS: merge5
    5
    PASS: split1
    -2 2
    PASS: split2
30  2 1 3
    FAIL: split3
    3 2 1 2 1 6
    FAIL: split4
    1 1 1 1
35  PASS: sort1
    1 2 2 3
    FAIL: sort2
    3 4 2 4
    FAIL: sort3
40  -5 -6 -2
    FAIL: sort4
    -6 -5 -2
    PASS: sort5
    testcase probe/testsuite/sorttest/mergesort.exp completed in 1 seconds
45
                    === sorttest Summary ===

    # of expected passes            9
    # of unexpected failures        5
50  runtest completed at Thu Jul 25 17:42:25 2002
```

All `merge` tests passed and also the first two `split` test cases, so it's very unlikely that `merge` contains errors. First problems occur in `split`, which should be examined further to find the bugs. Sometimes test cases can also help to identify the cause of errors, but normally test suites can't replace real debugging methodologies. Thus tests help to find bugs, but not to correct them.

5 Miscellaneous stuff

5.1 Own experiences

I was a little bit scared first, because I couldn't find many examples to work me into DejaGnu. At present the official documentation seems complex and is more a reference. It's a bit difficult to learn DejaGnu with it, because the function descriptions aren't ordered by their importance to the beginner.

But to be fair, DejaGnu really isn't very complicated. The syntax inherited from Tcl can be explained in ten lines and there are only a few commands, one has to know in order to use it. The documentation is also getting better every day.

In my view DejaGnu is really a good framework for general testing. Because it isn't bind to an graphical interface, one has the ability to use a big range of other tools greatly enhancing DejaGnu's flexibility. Also it isn't bound to a specific programming language or application programming interface, but allows specialization towards the needed features for the specific task.

5.2 File downloads

The whole package with all files and examples is freely available and newer versions will be published at my homepage (http://home.wtal.de/meph/). Contributions and corrections are very welcomed. Feel free to email me (R.Rotta@gmx.de).

5.3 Internet resources

http://c2.com/cgi/wiki?ExtremeProgramming	all about eXtreme Programming
http://c2.com/cgi/wiki?FunctionalTests	a few word about functional tests
http://c2.com/cgi/wiki?UnitTests	and about unit tests
http://c2.com/cgi/wiki?TestingFramework	overview about testing frameworks
http://www.scriptics.com	Tcl
http://expect.nist.gov	Expect
http://www.aptest.com/resources.html	a collection of many commercial tools
http://www.testingfaqs.org/tools.htm	a list with even more tools

Contents