Quality Systems

Creating Reliable Software

May 27, 2006

presented by

Made to Order Software Corporation
9275 Blue Oak Drive
Orangevale, CA 95662
U.S.A.

Tel: 916 988 1450

email: contact@m2osw.com

• Quality Systems

When do you consider Systems to be of good Quality?

Most people have come in contact, one way or another, with software and hardware: using your desktop computer, calling friends on your cell phone, watching a DVD...

Once in a while, the system breaks. The usual function or functions just do not respond anymore. The software or hardware *crashed*. This is called a bug in reference to a real bug which made one of our early computers malfunction.

So? How can you avoid system breakage? To avoid bugs, the best way is to use the system you are creating. Today, many companies will release software early to million of users and get direct feedback from their users. This works for application like desktop software. Of course, even in these companies, there will still be a *test period* (alpha, beta, release, gold master) before the software goes out.

Here we propose to demonstrate how a good and high quality testing scheme can help your company cope with new technology and its inherited bugs.

• Plan your tests

The best way to make sure that you will give enough time to your staff to fully test a new product is to incorporate the time it takes to test that product in your work plan.

From the start, when you analyze a new system, break up your project in small pieces (often called *modules* or *libraries* in software and *units* in hardware.) This process will help you determine:

- what will need to be tested
- how long it will take to produce the tests
- how long the tests will run for
- how much time your engineers will need to debug.

It is very important to account for the time it takes to run a test. For instance, when creating a complex hardware chip, you generally end-up with tests taking days to run to completion. If you find a bug which needs to be fixed, then you will need to re-run all the tests for several days!

Whenever possible, have staff create tests in parallel with the system. This gives you a chance to run the tests sooner and catch bugs sooner. Remember that the earlier you catch a bug, the fastest your development process is. It is much easier for an engineer to fix a bug in what he worked on the day before rather than 6 months ago.

• Software Libraries

If you are creating software which require libraries, you need to test all the functions which will be visible publicly once the library is released.

- Write a succinct definition of each public function
- Have one programmer develop the functions
- Possibly in parallel, have another programmer develop corresponding tests
- As soon as a function and corresponding test are deemed complete, run the test against that function
- If the test uncovers bugs, fix them as soon as possible

To create the highest quality software, you must test every single function. It is a daunting task, however, the results can be amazing. Note that for that purpose, the use of automated tools can greatly help.

• Hardware Units

Hardware Units are similar to a complex set of logical doors or transistors. A Unit cannot be broken up in smaller parts. On every clock tick, a Unit processes one byte of information (note that today, chips can support bytes of 256 bits.) The byte enters the Unit on one side and exists on the other via ports usually attached to small memory buffers called FIFO (First In, First Out).

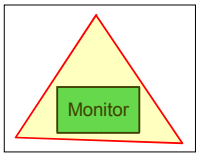
The process to create a Unit is as follow:

- Write a succinct specification of what the Unit does with its input and output ports
- Have one programmer develop a VHDL representation of the Unit
- Possibly in parallel, have another programmer write corresponding tests
- Debug the VHDL by running the test against it
- Draw the hardware design
- Test the hardware design against the test

Tools will be used to send the input of the test to the VHDL unit (and later to the hardware design) and then to compare their output.

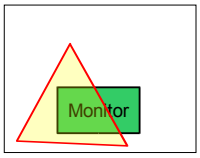
• Graphical Example

Here I show all the different tests you would need to run against a program drawing triangles to make sure that all cases are covered.



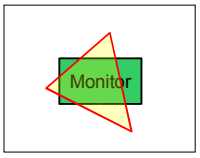
Case where the triangle is larger than the screen. The program will have to make sure that it clips and never writes outside the screen buffer. The test can include extremely large triangles to make sure that the program can handle triangles of any size.

Figure 1 – triangle larger than the output



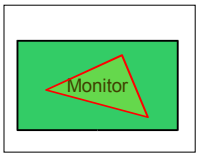
In this case, only one segment of the triangle is drawn on the screen. Again the program must make sure that the rendering does not happen outside of the screen buffer. The test can include cases where the other two lines match the edge of the monitor.

Figure 2 – triangle cuts monitor in two



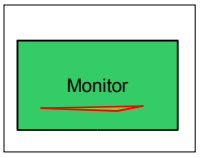
Yet another case where the triangle corners are outside of the screen buffer. The program is supposed to generate triangles to render the visible area. This case generates many of them.

Figure 3 – invisible corners



This is what we could call the common case. Most triangles will not be empty and they will be in the screen. The test should try with special case triangles (with a right angle, isoscele, equilateral)

Figure 4 – common case



Finally, you need to test very small triangles. Triangles of just 1 pixel height but very long, of 0 pixel height, with all points confounded. All the special triangles should be checked when larger and smaller than the screen and on the edges (inside and outside).

Figure 5 – flat triangles

For a library such as **OpenGL** you will also need to render two triangles right next to each others (as in a square). None of the pixels should be rendered more than once otherwise pixels partially transparency will not work correctly.

• Software Example

When writing a function in software, you need to test the validity by calling it from a test and testing the results. This usually means the tester will write a duplicate of the function to compute his/her own results (unless the results can easily be written by hand in a table). The test will then call the tester and the definitive function with specific parameters and compare the results.

Let's define a function computing a result of 0 when its input is zero, 1 when its input is odd and 2 when the input is even other than zero. The function is defined as:

```
function Oddities(value: Integer): Integer;
```

The tester writes his function in C:

```
int test_Oddities(int v)
{
      if(v == 0) return 0;
      if(v & 1) return 1;     // odd
      return 2;               // even
}
```

The test can look something like this:

```
int main(int argc, char *argv[])
{
      int i, err = 0;
      for(i = -10; i <= 10; i++)
      {
            if(Oddities(i) != test_Oddities(i))
            {
                  fprintf(stderr, "ERROR at %d\n", i);
                  err = 1;
            }
      }
      exit(err);
}
```

As we can see, the test will check negative and positive values and zero which is a special case. This should cover all the possibilities thought this test is not covering all the possible values, it is likely to be enough. To make this test stronger, you could test with values around INT_MAX and INT_MIN. Note that if you have the time, you should call such functions with all the possible values.

• Hardware Example

When creating a chip, you must test every single Unit. This is very important especially if you are to fabricate millions of them.

The best way to test a Unit would be to send all the possible values to its input port and see that its output port reacts according to the specification. However, now that the input port can be 256 bits, you just never have the time to test every single value. It would take years (2^{256} is quite large!) So what you want to test are all the specific values which the Unit understands and tweaks, plus some random values which the Unit is supposed to either block or pass through.

Let's say we are working on a Unit which takes a 256 bits value. In these 256 bits, the lower 4 bits indicate some command. If that command is 6, then it has to make sure the other 252 bits represent a value between 0 and 1023 (in other words, the following 10 bits can be either 0 or 1, and all the other bits will be cleared.)

To write a test for such a Unit, you want to have a way to generate pseudo random values with all the commands other than 6. You can write a loop which goes from 0 to 15 (the command is on 4 bits so the maximum command number is 2^4 - 1):

```
for(i = 0; i < 16; i++)
{
      if(i == 6) continue;
      input = random(); // generates a random number of 256 bits
      input &= ~0xF;    // clear the command bits
      input |= i;       // set the bits to i
      print("in: %d, out: %d\n", input, input);
}
```

Note that in this case the input and output are the same.

Now, in case of command 6 we want to randomize the 242 bits which need to be cleared by the Unit. If any one bit from these 242 aren't cleared, then there is a bug. You should repeat the test sufficiently so all the bits are tested multiple times with 0s and 1s. You also need special cases where all the bits are set and all the bits are already cleared. For good hardware tests, it is also a good idea to test bit patterns (10101010 or 0xAA for instance.) So the loop could look something like this:

```
char pattern[5] = { 0, 0x55, 0xaa, 0xff, 0x01 };
for(i = 0; i < 5; i++)
{
      for(j = 0; j < 10000; j++) {
            input = random();
            if(pattern[i] != 1) {
                  // write the pattern at position (j % 32)
                  input = set(pattern[i], j % 32);
            }
            // force the command to 6
            input &= -16;
            input |= 6;
            output = input & ~0x3FFF;
            printf("in: %d, out: %d\n", input, output);
      }
}
```

In this example, we clearly see that the output has 242 bits cleared. The hardware, once working properly will do the same thing.

● Need help with your test suite?

Made to Order Software Corporation helps companies create their test suites from the start. Hardware or Software, we will help you accomplish what you want to achieve. We can write the tests for you or help you decide which automated software will help you reach your goal faster.

We are located in California, USA and we also have an offshore office in London, UK.

Please contact us by phone at +1 (916) 988 1450 or by email at contact@m2osw.com .