In-vivo project

In this report, I briefly introduce the structure of in-vivo and what functions each file implement.

1. How it works:

(Files are in the invivoCompiler folder)

\* The segment of code you want to test (demo.c)

\* create a configuration file (demoTestInsert.ic) that specifies the signatures of the methods for which you want to insert tests, and then the code of the tests themselves, along with the probability with which the test should run.

\* run the "invivo compiler" (precompile.pl) with the two other files as arguments. It will convert demo.c to output.c which will contain all the instrumented code. (./precompile.pl demo.c output.c)

Then compile your code as normal, using the newly generated code (output.c) as well as InVivoController.c, which contains some helper functions.

1. How to transfer demo.c into output.c

Let's say you have a piece of code and you have some function like this:

int add(int x, int y) {

return x + y;

}

And you want to execute the invivo test(s) for that function. What you would need is to rewrite the existing code into something like this:

// this is the original code, but renamed

int \_add(int x, int y) {

return x + y;

}

// this now wraps the original

int add(int x, int y) {

// see if we should run a test

if (should\_run\_test("add")) {

// if so, fork

if (fork() == 0) {

// this is the child... run a test

test\_add(x, y);

exit(0);

}

}

// if we don't run a test, or if this

// is the parent after the fork, just

// call the original function

return \_add(x, y);

}

So now whenever "add" is called, it determines whether an invivo test should run. If so, it forks a process that runs that test. And then, no matter what, it calls the original implementation, which has been renamed "\_add".

The should\_run\_test function is in InVivoController.c file and has some intelligence to decide whether to run a test. This is based on a random coin flip, or some measurement of system load, or anything else.

The test\_add function would be defined either in this .c file or maybe some other file that contains all your tests.

Now when you compile everything using GCC or other compile tools, the code has been instrumented so when "add" is executed (via some test drivers that force it to be called... note that those test drivers are not part of the testing framework) it may invoke its test function in a separate process.

So now what you need is a way of converting the original version of "add" to the instrumented version, so that the developer doesn't have to do everything manually. That was the intent of the the invivoCompiler. When you run

./precompile.pl original.c tests.ic

where original.c is your original source code and tests.ic is a configuration file that would contain lines like this:

int add(int x, int y) pre

void test\_add(int x, int y) .5

{

if (\_add(x, y) != x + y) printf("test failed!\n");

}

The first line of the configuration file says "there is a function with this signature, and the invivo compiler should create a pre-test for it, i.e. a test that runs before the function is actually called".

The second part is your actual test function. Note that the .5 afterward indicates the probability that the test will run, and that in the body of the code, if you want to call the function (which you presumably do), you need to call "\_add", not "add", otherwise you'll end up in an infinite loop.

If precompile.pl runs correctly, it will create a file called output.c, which is original.c but with the test instrumentation added. Now you would just run your regular C compiler with output.c (instead of original.c) and InVivoController.c, which has the should\_run\_test function and some other stuff you'll need. Last, you would need to write some other test drivers that would invoke "add", and when it's called it will launch its invivo tests.

If precompile.pl doesn't work, you can add the instrumentation by hand, just by adding the should\_run\_test and fork calls manually to the functions that you want to have tests for.