Invite Research Project

Fall 2009- Final Report

Waseem Ilahi – wki2001@columbia.edu

Professor Gail Kaiser

Chris Murphy

Abstract

There is a limit to how many tests one can perform on a software before deploying it. This limitation makes it possible for some of the bugs in the code to go unchecked. Thus, the system/software is released with those bugs still present and unknown to user and developer. Our research is based on solving this problem in the manner that even after the software has been released to the general public; there can be a way to keep testing it continuously, without disrupting the original order of execution. This functionality is achieved by forking off the original path and performing the test in the child process and the continuing the original path. Doing so makes sure the original path is not mutated. Some virtualization method can be used to make sure the original process isn't affected. Invite is the "system" that implements this continuous testing approach. We call it InVivo testing approach. The original executed these tests whenever it came across them, thus executing many of them various times. This is inefficient, so we need to find a way to avoid this problem. We can specify a certain "state" that a process has, right before it executes a test. These states can be hashed into a data structure that allows very fast iteration through it. So, when a process reaches at the beginning of a test, it can look into the structure and find out if the "test" has been executed before. If it has, then we can simply ignore the test and move on, otherwise we execute the test.

The "state" in this case consists on the variables that the test will depend upon. These variables can be found by the static analysis of the test code. I was given the job to implement this "analyzer/preprocessor" that would figure out these variables and return them.

This paper provides an introduction of the whole project and then the description of the code that I wrote. In this paper, I will try to include a lot of description for my work in Fall 09, so the future project students can use this paper to continue from where I left off.

1. Introduction

The traditional testing is never able to find all the bugs in the program. PSL has been trying to investigate “InVivo Testing Approach” to solve this problem. This method performs the tests continuously, even after the deployment of the software. The functionality that we had so far, for this suite was that it forked the original process and then the test was run on this child thread. This process leads the system to perform a test multiple times, especially in distributed systems. This is a huge performance hit. To avoid this, Mr. Chris Murphy came up with the idea, that we create some sort of a “state” and hash it, so when the process reaches the execution point of the test, it checks the hash and if the test has been performed before, it simply continues without performing the test, otherwise, it performs the test and adds this “state” in the hash for the future. This is a two step process, first is to figure out what the “state” is and next is to use some “data structure” to “hash” these states in it, for fast access. We decided that the “state” will be the values of all the known external variables that a test depends upon, at the beginning of the test. My responsibility this semester was to find these variables. I had to implement a preprocessor that takes a “test/function” and gives back the variables, this test depends upon.

This paper describes the code that I wrote that implements this functionality. Section 2 gives a little insight into the “big picture” of the whole project. Section 3 talks about what I had to implement. Section 4 explains the various parts of the implementation of my code. Section 5 gives the list of files and a little description of what each contains. Section 6 is about future directions. Section 7 will talk about lessons learned and finally Section 8 is references.

2. Big Picture

As mentioned above, Invite takes a performance hit, we a test is executed multiple times. In this regard the following question was asked, “Can the technique be made more efficient by only running tests in application states it hasn't seen before? We currently run tests probabilistically or according to the system load; however, certain states may unnecessarily be tested multiple times.”[Chris Murphy (proposed topic for research)].

To actually test this theory, we needed to create a state representation and then hash these states (visited) into a data structure that accesses them very fast. This task has two parts, first is to find that state representation and second is to find that data structure that lets us do what we want. The first part was assigned to me and the second to my friend Moses.

The coming sections talk about what I did, so here I will talk a little about the second part. After we have a “state” we can simply hash it to a data structure (Judy Array). So before the test is run, we can ask the question, whether the state is in the “Judy Array” or not, if it is, we don’t run the test, and if it is not, then we add it to the array and run the test. Following is a snippet of code that represents the change of function “f” into what we want.

Original Code:

```
int k;

int f(int a , int b){
    return a + b + k;
}
```

State-Aware Invite Generated Code:

```
int k;

int _f(int a , int b){
    return a + b + k;
}

int f(int a , int b){
    if(runtest_f(a, b, k)){
        int pid = fork();
        if(pid == 0){
            test_f(a, b);
        }
    }
    return _f(a, b);
}
```

Now, my code figures out the variables “f” depends upon and then passes them along to “runtest_f” which accesses the “Judy array and decides whether to run the test or not.

3. Preprocessor

From the snippets of code above, we can see that “_f” relies on “a,b,k” , so magically “runtest_f” has the three variables as its arguments. In actuality, it is the job of my “preprocessor” to figure out these variables and then hand them to another “preprocessor” that creates the above shown “Invite-generated code”. This preprocessor has its limitations. As it is, by the end of this semester, it works for “compliable (no errors)” c code. It covers basically any data structure, even arrays and pointers. However, it is not equipped to handle some of the keywords, e.g., volatile, etc.

One of the major steps in implementing this was to handle re-declarations of the variables and setting the dependencies according to that. However, I did manage to complete it. Following section will go in detail of this processor.

4. The CSInvite preprocessor and Integration with InVite { CSInvite.exe }

After I was done with the research on finding a “preprocessor” that figured out the dependency of a function on the external variables, without success; I started programming for the preprocessor myself.

The preprocessor “CSInvite” is run from the command line and it accepts one argument; a “c” source file. For Example, on a linux machine one can type “./CSInvite c_file.c” on the command line to run the preprocessor. Now I will talk about what the processor does and how the user gets what “he/she” wants from the processor.

When the preprocessor is run, it loads up the file and reads it. In the background, hidden from the user, it reads and stores the “Global Variables” and “Global Constants” in the file. It also divides the file into separate function definitions. If there are none, then it lets the user know and exits. Now that it has everything ready, it asks the user to input the name of the function they want to know the dependencies for.

The user types in name, if there is no name in the “database” that matches what the user typed, then the “preprocessor” tells the user that there is no function with the name the user typed and lets the user type again. If the user types “quit”, the processor exits and the control is returned to the shell. Now, if the user types in the correct name of one of the functions in the “database”, the preprocessor goes in and gets the function definition for that function. It divides this definition into separate statements. These statements are scoped, relative to the function definition not the entire file. The scope is an int that starts with the number “0” for the highest scope, going to numbers 1, 2, 3 for smaller scopes relatively.

After we have the statements with scopes, the preprocessor calls another function that sets up the “Local Variable” data structure. These aren’t all the local variables; they are the only ones that are the re-declarations of the “global variables” and parameters.

Now that we have these variables, preprocessor takes each “Scoped Statement” for the current function (set in the previous step), and finds the variables that can affect the function’s behavior. These resulting “values” are not necessarily all variables, they can be “numbers”, function names, etc. To make sure we only get the external variables, the preprocessor calls another function, this function takes each of these “values” and compares them against the “Global Variables”, “Global Constants” and “Parameters”. If it’s any of these then it gets assigned to the structure that holds the “Final Variables” that the function depends on. The last thing that this function checks for is that whether this “value” is another function name, especially one of the functions, defined in the same file as the original function that is being tested. If it is, then it is tested to make sure, that function has not been “traversed” before (to avoid infinite recursion). If it has been traversed before, we move on the next value, if not then another function is called, this function repeats all the steps, from setting the scopes for this new function that is being traversed to getting the local variables and setting the dependencies. It then calls the “set_variable” function, which called it. When that function “set_variables” returns, this function returns. And then the process continues, until all the values have been checked. The control is then returned to the preprocessor.

The preprocessor has the variables the function, that the user asked for, depends upon. It does one last thing; it removes the duplicates, from the array. Then it prints out the variables and their types to the screen. If there were no such variables, then it says that the function doesn’t depend on any external variables. It then prompts the user to input another function name to test

its dependencies. This is the process that it goes through in figuring out the function's dependencies. Following in the list of the function, with brief description, that the preprocessor calls (mentioned above).

1. int get_input(char (*statements)[MAX_LENGTH], char *filename);

   This function gets the input from the file and copies it to the "statements" array. Each element of "statements" is, either a global variable/constant, or the function declaration.

2. int get_possible_constants(char (*statements)[MAX_LENGTH], int statement_number);

   This function simply returns the possible constants in a file.

3. int set_global_constants(char (*statements)[MAX_LENGTH],
   char (*global_constants)[MAX_LENGTH], int statement_number);

   This function sets the global constants in the file and returns the total number of constants.

4. int set_global_variables(char (*statements)[MAX_LENGTH],
   GlobalVar global_variables[], int statement_number);

   This function sets "GlobalVariables" for the current "source" and returns the total number of variables.

5. int set_functions(char (*statements)[MAX_LENGTH],Functions function_list[],
   int statement_number);

   This function sets the functions names and their definitions, so they can be used when the user asks for them. It returns the total number of functions that were defined.

The following functions are called when the user gives the name of the function.

6. int find_function(int function_number,char *funcs, Functions function_list[]);

   It simply checks whether the function definition exists in the file provided. If yes then it returns non-negative value, otherwise it returns negative.

7. int set_parameters(char *definition, Parameter parameters[]);

   Now, that we have the correct function name, that is present in the “database”, we can get its parameter. This function does just that.

Now the following functions are called each time, a function is “traversed” (even for a function called inside the function).

8. int set_statement_scopes(char * definition,
   Scoped_Statements function_scoped_statements[]);

   This function gets the “traversed” function’s definition and breaks into scoped statements. Then it returns the total number of these statements.

9. int set_declared_local_variables(Scoped_Statements function_scoped_statements[],
   int total_scoped_statements, LocalVar declared_local_variables[],
   Parameter parameters[],int total_params, GlobalVar global_variables[],
   int total_globals, char (*global_constants)[MAX_LENGTH], int total_constants);

   This function sets the "Local Variables" for the current function being "traversed". These "Local Variables" are the re-declarations of the parameters and global variables. It returns the total number of "Local Variables" found.

10. int set_dependency(int total_dependent_variables,char *statement,int scope,int number,
   int total_local_variables, LocalVar local_variables[],
   char (*dependent_variables)[VAR_LENGTH],
   Functions function_list[],Parameter parameters[],
   GlobalVar global_variables[],
   char (*global_constants)[MAX_LENGTH]);

   This function sets the "dependent_variables" array, based on the variables that affect the execution of the function/test. This is where, the re-declared variables, inside the scope are "ignored".

P – T – O →

11. int set_variables(char (*done_func)[VAR_LENGTH], int total_done,
    int total_variables,int function_number, Functions function_list[],
    int total_params,Parameter parameters[],int total_globals,
    GlobalVar global_variables[],int total_constants,
    char (*global_constants)[MAX_LENGTH],
    int total_dependent_variables,
    char (*dependent_variables)[VAR_LENGTH],Variable variables[]);

This function goes through all the "dependent_variables", compares them to the parameters, global variables and constants and the list of un traversed functions, and acts accordingly. If it is one of the "variables/constants", then it is added to the array. If it the last possibility, i.e., non-traversed function, then it calls the "get_func_vars", which repeats the whole process for that new function. "set_function" also "flags" the "non-traversed" function as traversed.

12. int get_func_vars(char (*done_func)[VAR_LENGTH], int total_done,char *func,
    int function_number,int total_globals,int total_constants,char *definition,
    int total_dependent_variables,
    char (*dependent_variables)[VAR_LENGTH],int total_variables,
    Variable variables[],Functions function_list[],
    GlobalVar global_variables[],char (*global_constants)[MAX_LENGTH]);

This function, as mentioned for the previous function, repeats the whole process for the "non-traversed" function, sets the "Variable" structure and returns the "Variables" found in "non-traversed" function.

Integration with InVite

As I have mentioned before, even though the preprocessor works great on its own, it hasn't been integrated with the Invite code yet. We need to implement another preprocessor that takes the original code, gets the variables from this preprocessor and outputs the final code. Something like the code snippet provided above. For the purpose of testing, we create the final code manually. We take the variables gotten from the test and manually put them as "runtest_f's" arguments.

The code is commented itself and very self explanatory, what is important to mention here is the steps to take for the whole process. I do provide the steps for a small demonstration, using the sample test code that I provide with the source.

Demonstration

The test code provided is in the "Test" folder. I will go over the steps for the file named "single_test.c" inside the "Test" folder.

i) cd /home/CSInvite/ /* move to the directory of the source code. */

ii) make /* makes the target "CSInvite.exe" */

iii) ./CSInvite single_test.c

iv) first , second , third , fourth , fifth , sixth , seventh → these are the choice you have, type any one or all separated by the space. You will see the results printed on the screen.

v) Type "quit" for exiting the program.

I created the program this way, because it has not been integrated with the other components. Minor changes are required to integrate this preprocessor with the whole project.

To further understand the functionality, read the source code, provided with this final report and also look at the Section 5 below. The material in Section 5 is also available as a README with the source.

5. The Source Code:

My goal for the semester was to write a "preprocessor" that takes a "c" file reads it and for each function defined in that file, if finds out the variables that this function depends upon. This functionality is very important for the main project that we were working on, which is to figure out the impact on the performance on the Invite system, if it knows whether it has seen the current state or not and if it has, then it skips the test. My code comes into play, when we talk about a state. We decided to differentiate among the states, based on the values for each variable, the current test depends upon. My code finds out these variables and after the integration, will pass along these variables to the next "preprocessor that creates the resulting code for testing.

I divided the code in multiple files with each file housing the code called on the similar levels. I am listing the files and the functions in each with a little explanation for each.

The List of Files and Directories:

a. The CSInvite directory

CSInvite.h → It is the main header file for all the components. It has the "includes" from the c library. It also contains the macros for the lengths and numbers. Then it defines the data structures for all the variables used by the program. And at the end, it has the prototypes of all the functions used by the program.

CSInvite.c → The main file. It calls all the other functions and does controls the program flow. It contains the main function.

CSInvite_help.c → This file holds the three string helper functions. These functions are called by other functions to do string manipulation.

CSInvite_set_input.c → This file holds the functions that are called by the main function once after the invocation of the program. It sets up the global variables and constants, and the functions with their definitions.

CSInvite_set_func.c → This file sets up the scopes of the statements, plus the re-declared variables for each function, the user asks for.

CSInvite_set_vars.c → This file contains the functions that actually fill the structure that represents the variables the function depends upon. It also contains the function to handle calls to other functions and recursion.

Makefile → makes CSInvite.exe (can be used in Windows or Linux)

b. The Test Directory

This directory contains the test files. These tests were used to check the functionality of the program.

test.h, test.c, test1.c and test2.c → These are the basic test files, test1.c and test2.c have one function definition and test.c calls these functions.

single_test.c → This file contains seven function definitions, these functions test various features the program provides.

Makefile → makes the test.exe, also checks single_test.c for problems.

NOTE: "compile" the test code before using it with the "CSInvite", for the program to work correctly.

6. Future Direction

We have the preprocessor to get the variables the function depends upon; we also have a way of checking whether the state was visited by the test before or not. Next step is integrating both features and taking care of few limitations they each have. We might also want to try and include some external variables to expand the “definition” of state we have. Right now it is pretty limited; we will want to expand that to include external variables as well. We did achieve good numbers while evaluating our technique, so there is definitely some potential in going in this direction.

7. Lessons Learned

First, I had to researh whether there was something already present, like what I had to do. During the extensive research on the subject, I came across a lot of good technologies/tools, one of them being “Frama-C”. I learned a lot about the static analyzers. However, none of them did what we were looking for. Therefore, I had to implement the “preprocessor” myself. Since, it is something totally new, I didn’t have many references to consult. Working with “c” gets complicated, but I enjoy working with it. I was able to under the whole syntax of c programs much better than before. I used many new functions provided by c standard library. It was fun, but it was very complex and took a lot of my time.

Nevertheless, I am pleased that I was given the opportunity to work with this project. I give my thanks to my project advisors Mr. Chris Murphy and Dr. Gail Kaiser, for providing me the opportunity to work on such an interesting subject, my contribution made me feel proud and satisfied. I am planning to go far the PhD. I hope to work on similar projects in the future if possible.

8. References:

Google Search Engine [Multiple Forums]

Frama-C Documentation. [http://frama-c.cea.fr/]

Waseem Ilahi (myself) [Summer 09 Final Report]

The Entire "Project" (code, proposal, final report, tests) can be "downloaded" from the Google code repository, from the URL: http://code.google.com/p/clever-state-invite/