The Report about Measuring Coverage on Correct Software or a Fault One

Mengyuan Wang
Carleton ID:101289282
Email: mwang259@uottawa.ca
Supervised by: Prof. Yvan Labiche
Department of Electrical and Computer Engineering
University of Ottawa, ON, Canada, K1N 6N5

***Abstract*—This paper is a software testing task for project 1: Measuring coverage on correct software or a faulty one. project1 is an object based on c language. This report selects four subject programs in the SIR dataset library, namely tcas, schedule2, replace and totinfo. For the correct version of each program and all the fault versions generated by implanting seeds, perform all statements and all branches coverage, and which test case reveals the collection of fault detection information the fault and the relationship between the two and the relationship with test case size. Finally, a data matrix of 1 and 0 is generated, and the project finally uses the generated matrix for processing, and randomly selects a test set to test the program chosen to obtain a scatter curve of the relationship between coverage and fault detection when the coverage is from 0 to 100%. Trend graphs of fault detection and coverage as test case size increases. Here, two different methods were tried when generating the fault matrix, and corresponding scripts were written using python languages to realize automated testing and improve testing speed, especially for programs with many versions and many test cases. The tedious running time of manual operation is significantly reduced.**

***Index Terms*—software testing, fault detection, gcov, coverage**

I. Introduction

All the programs selected in this report come from the data set provided on the SIR website to complete the software test experiment report. SIR——Software-artifact Infrastructure Repository is a very famous data set in the field of software research.

From the official website of SIR, it can be seen that the subject programs selected by this paper belong to the Siemens data set, and the test set contains 7 programs in total. Each program corresponds to a correct version and multiple versions with bugs and contains several test cases. These programs are all written in C language, but because the test set has a long history, many of its C language grammars do not conform to the current GCC compilation specification, so some codes need to be manually modified.

After decompressing the program, you will find a universe file under testplans.alt. This is because the assembly selected in this article is an old version. The main difference between the new version and the old version is that the test cases of the new version are based on ".universe" The file with the suffix is saved in the format of the mts tool; while the old version of the test case is to save the parameters of the test program line by line in the "universe" file (not in the format of the mts tool). When using it, you need to convert the universe test script to mts format ".universe" through sed first. The following program schedule2 will explain it in detail.

The Fig1 shows that the basic information of subject programs selected:

Program	Lines of Code	Origin version	Faulty versions	Test Cases
tcas	173	1	41	1608
tot_info	565	1	23	1052
replace	564	1	32	5542
schedule2	374	1	10	2710

Fig. 1. Basic information of subject programs selected.

The equipment and systems used for the tests were as follows Fig.2:

CPU	Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz
Memory Total	32699648 kB
OS	Ubuntu 7.5.0-3ubuntu1~18.04

Fig. 2. The information of the system.

My root path shows in Fig.3.

experiment_root	export CLASSPATH=$CLASSPATH:"/home/whp/Documents/project/"

Fig. 3. Root directory environment variables.

The next step is to download and configure the environment for the Mts tool that needs to be used. mts is short for MakeTestScript, in the mts directory, there is mts.jar under the bin folder and antlr-2.7.6.jar under the lib folder, add these two jar packages to the CLASSPATH and add bsh or csh from the bin directory to the PATH. As shown in Fig.4.

There is another very important step. Set the MTS_PATH in the two script files under bin/bash to your own path, for example, mine is set to MTS_PATH = "/home/whp/Documents/project/mts". In addition to the tool mts, the test also uses gcov, a C/C++ code coverage analysis tool that comes

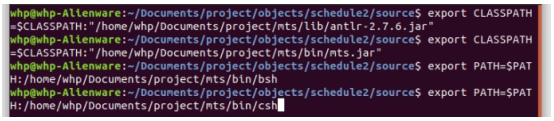

Fig. 4. Mts tool environment configuration.

with GCC for Linux. For coverage collection, we use GCC compilation. The GCC environment is installed as follows:

sudo apt-get update

sudo apt-get install gcov

At this point, we have a general understanding of the program we want to test and a test environment ready for the next step.

II. Overview of program testing

1. Select the version of the program to be executed, and move it to the source folder, because of the default execution of the ".exe" program in the runall.sh test script under scripts is in the source directory, and the default output is in the outputs folder. If we want to put the test script and the execution program in the same directory, we can change the ".exe" path and output path in runall.sh to make the subsequent series of operations clearer and more convenient. These two operations in different directories have been verified to show that the results are the same.

2. GCC compiles and generates ".exe" program files: add -fprofile-arcs -ftest-coverage options.

3. Execute the test script.

4. Execute gcov tcas.c / gcov -b tcas.c to generate a tcas.c.gcov file.

5. Execute gcov-compiler to generate CSV matrix file.

6. Run the python script or execute the compare test script for the fault location.

7. Randomly select test cases to form test scripts according to the generation matrix, and execute them separately to see how the coverage and fault detection will change with the increase of the use case size.

III. Detailed Test Procedures

A. Detailed test of tcas

Here I take tcas as an example to introduce how to test in detail, and the other three procedures are very similar to tcas. After decompressing the project, the directory structure contains: info, inputs, outputs, newoutputs, scripts, source, source.alt, teatplans, testplans.alt, traces, versions, versions.alt.

All test case inputs are saved in the Inputs directory, and all test case inputs are provided in the form of stdin. But tcas is special, because the test case input of tcas is relatively simple, only a set of digital input, so the test case input of tcas is directly written into the test script, therefore, the inputs directory in tcas is empty. The test case inputs of other sub-suites are saved in the input directory of this sub-suite in the form of text.

The Outputs directory is used to save the correct prototype of the program under test (that is, the code in the source.alt directory) for the correct output given by each test case, and is used to compare the execution results obtained by the same test case when testing a version containing errors same. The outputs directory is temporarily empty until any test program is executed.

The Newoutputs directory is used to save the actual output of each error version of the program under test for each test case and is used to compare whether the execution results of the same test case are the same when testing a version containing errors. Before any test program is executed, the newoutputs directory is temporarily empty.

The test script example given by the sub-test suite is saved in the Scripts directory. After opening the scripts directory, you can see that there is only one shell script file named "runall.sh" in this directory, which is used to run the tested program in the source directory. After inputting the test cases into the correct prototype of the program under test, output the execution results to the outputs directory in tcas, and generate the files according to each test case. "runall.sh" defines 1608 test cases in total, that is, the total number of tcas sub-suites 1608 test cases. Enter the command "./runall.sh" in the scripts directory to run the script file. Before running the script, please compile the "tcas.c" file in the source.

First, copy the tested program tcas.c under source.alt/source.orig to source, and compile it into a binary file named "tacs.exe" in advance for the test program to call. Other sub-kits may have several code files/header files, but in the end, only one binary file is generated to execute the program. Use the standard GCC compiler on the Linux platform, and use the command "gcc tcas.c -o tcas.exe" to compile the tested code. Note that "-o" is a lowercase letter "o", indicating the specified output file name. In the actual test, it is not enough to know whether the program under test is running correctly or not. It is also necessary to obtain the running information of the program, that is, in the next run, which codes are executed, which codes are not executed, and each line of code is executed how many times, this is the desired detail. In order to obtain this execution information accurately, we need to use a tool called GCOV, which is integrated into the GCC compiler to count the execution information of the compiled program. To use GCOV, you must first add the two compilation options "-fprofile-arcs -ftest-coverage" when compiling the program, that is, use the command: "gcc -fprofile-arcs -ftest-coverage tcas.c -o tcas.exe" to compile tcas.c code file, where:

* -ftest-coverage: A .gcno file is generated at compile time, which contains the line number information of the reconstructed basic block graph and the source code of the corresponding block.

* -fprofile-arcs: When running the compiled program, a .gcda file will be generated, which contains information such as the number of arc jumps.

Next, we enter the scripts directory and run the "./runall.sh" script file, and input the test case into the program. After execution, we will get a "tcas.gcda" file in the source, and then we can use "gcov tcas.c" normally to get coverage

information for the entire program. But this only covers the coverage of code lines executed without branch coverage. If you want to obtain more detailed branch and function call coverage information at the same time, you can add the "-b" parameter (this parameter outputs branch statement frequency information to the output file, and summary information to stdout, but does not show unconditional branches) ie: "gcov -b tcas.c". The output comparison of the two is shown in the Fig.5 below:

```
whp@whp-Allenware:~/Documents/project/objects/tcas_2.0/tcas/source$ gcov tcas.c
File 'tcas.c'
Lines executed:89.83% of 59
Creating 'tcas.c.gcov'

File '/usr/include/x86_64-linux-gnu/bits/stdio2.h'
Lines executed:100.00% of 1
Creating 'stdio2.h.gcov'

whp@whp-Allenware:~/Documents/project/objects/tcas_2.0/tcas/source$ gcov -b tcas
.c
File 'tcas.c'
Lines executed:89.83% of 59
Branches executed:96.77% of 62
Taken at least once:91.94% of 62
Calls executed:100.00% of 13
Creating 'tcas.c.gcov'
```

Fig. 5. Detailed coverage information.

Enter the outputs directory after running the "runall.sh" script, we can see that the execution results of 1608 test cases are stored in the outputs directory in the form of text files, and these execution result files will be used for further fault location test use. The above is the modification and operation of a correct version of the tcas program, and the corresponding data is produced.

Now start the detection of the faulty version, this report only describes the first faulty version. The relevant data below are all from faulty version 1. For all the data of the other 40 faulty versions, please refer to the attachment. Each error version of the tested program is saved in the versions.alt directory, that is, in the correct program, several errors are randomly implanted, and the generated error codes are for everyone to test. The tcas sub-suite has a total of 41 error versions for everyone to test. Each version of the 41 error versions has one or more errors, but the errors are different between versions. Before testing these 41 wrong versions, we also need to pre-compile the code files into binary files, please follow the correct version above steps. Next, taking v1 as an example, collect the coverage information of the faulty version and compare the results with the output of the correct version.

In the previous introduction, we learned that the test script samples stored in the scripts directory are used to run the correct prototype of the program under test in the source directory. However, it is impossible to only run the correct program, inputting the test case into the fault versions is more important for the program to obtain and analyze its running results. The following will learn how to use the sample scripts given in the kit to get the test script corresponding to the faulty version. Open "runall.sh" in the scripts directory, we can see that the work of this script program is actually very simple, that is, to traverse the test cases, input each test case into the program under test in turn, and specify the result output file, the script's we have seen the running results above. To use the script to execute the fault version of the program is actually to modify the path of the executable program and the path of the program output. Here, for the convenience of execution and viewing, copy the script to each faulty version and modify the corresponding path. Taking v1 as an example, the modification is shown in Fig.6 .

```
echo script type: R
echo ">>>>>>>>>running test 1"
./tcas.exe  958 1 1 2597  574 4253 0  399  400 0 0 1     > ../../../newoutputs/v1/t1
echo ">>>>>>>>>running test 2"
./tcas.exe  627 0 0  621  216  382 1  400  641 1 1 0     > ../../../newoutputs/v1/t2
echo ">>>>>>>>>running test 3"
./tcas.exe  549 1 1 4398  133 1445 1  641  639 0 0 1     > ../../../newoutputs/v1/t3
echo ">>>>>>>>>running test 4"
```

Fig. 6. New script for v1.

From the generated new script, you can see that the "tcas.exe" in the current folder is executed, the middle number is the input of the test case, and the results t1, t2, and t3 are output to the v1 directory under newoutputs folder to avoid the name of each result is the same. For the sake of a clear follow-up operation, execute in the newoutputs folder: "mkdir v1..41" to create 41 folders corresponding to the faulty version number at one time, such as v1, v2...v41. And because there are 1608 test cases, it is not realistic to manually execute each test case. A python script "read.py" is written below to automatically loop through each test case, part of codes as shown in Fig.7(please check the attachments for the whole python script) .

```
with open('runall_v1.sh','r') as f:
    lines = [line.rstrip() for line in f]
l = len(lines)
counter = 1
with open('file_out.log', 'w') as f1:
    pass
with open('file_err.log', 'w') as f2:
    pass
fdout = open('file_out.log', 'a')
fderr = open('file_err.log', 'a')
fdout
for i in range(1,l,2):
    cmd0 = 'rm tcas.c.gcov tcas.exe tcas.gcda tcas.gcno'
    subprocess.Popen(cmd0, shell=True)
    time.sleep(1)
    cmd1 = 'gcc -fprofile-arcs -ftest-coverage tcas.c -o tcas.exe'
    subprocess.Popen(cmd1, shell=True)
```

Fig. 7. Part of the code of the read.py script.

This python script first reads the script file, two lines at a time, to obtain the test cases to be executed. Start to execute related commands, first check whether there are "tcas.exe", "tcas.gcno", "tcas.gcda" and "tcas.c.gcov" files in the folder, if they exist, delete them first, otherwise, the obtained coverage is cumulative. Next, operate as the correct version above, first, compile it into a binary file named "tacs.exe" in advance, two new files "tcas.exe" and "tcas.gcno" will be generated, and then execute the script: "./runall_v1.sh", and tcas will be generated "tcas.gcda" file. Use "gcov -b -c tcas.c" (-c is a number rather than a percentage to display the branch frequency) to generate "tcas.c.gcov" to the folder named after the serial number of each test case (1608), because each "tcas.c.gcov" file names generated by the test cases are all the same. In addition, the coverage of each test case executed by "read.py" will be recorded in the "file_out.log" file. Use "generate.py" to compile the "tcas.c.gcov" file generated by each test case, it will generate two files "tcas.c.gcov-Statements.csv" and "tcas.c.gcov-Branches.csv". At this time, there will be three files "tcas.c.gcov", "tcas.c.gcov-Statements.csv" and

"tcas.c.gcov-Branches.csv" under each file named after the serial number of the test case. Separately, they are shown in Fig.8, Fig.9 and Fig.10 below.

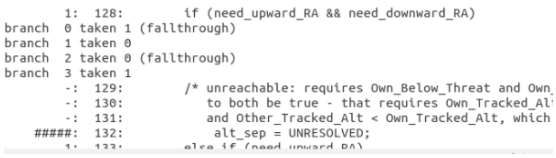

Fig. 8. The content of tcas.c.gcov.

In "tcas.c.gcov":
−: indicates that the code is not executable;
Number indicates the number of times it was executed;
###: indicates that the code was not executed.

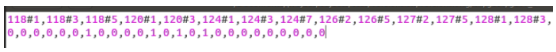

Fig. 9. Part of tcas.c.gcov-Branches.csv.

This is a record of the branch execution of a case. You can see that branch 3 of line 128 in Fig.8 was executed once and recorded as 1, so 118#3 of the matrix information is recorded as 1. The rest are all recorded as 0. However, because the branch coverage of each case is different, in order to maintain the uniformity of the total branch matrix length of each version, all the branch rows are extracted, and all the branch rows containing 1 or greater than 1 indicate that they have been executed, which is recorded as 1, otherwise 0.

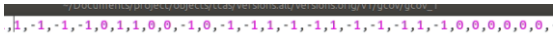

Fig. 10. Part of tcas.c.gcov-Statements.csv.

This matrix records the coverage of the statements of a case. The code is executed and how many times it is executed. Here it is recorded as 1, otherwise, it is 0. -1 means that the code cannot be executed, because it is the output of bin compilation, so retain the original -1 data. Finally, use the combinebranches.py and combinestatements.py scripts to combine the single-row matrix of 1608 test cases into a 1608-row CSV file (see the attachment for details). The coverage CSV for all statements and all branches for each release has been done.

Regarding the collection of fault detection information, after outputting the results of each wrong version to newoutputs, use the diff.py python script to compare the results with the correct version, the same is 0, and the difference is 1 to generate a result containing only 0 and 1 matrices "result.txt", and finally combine the 41 versions of the matrix into a combinefault.csv file through the combine.py script. part as shown in Fig.11.

Fig. 11. Part of combinfault.csv.

At this point, the fault detection matrix work of the tcas program has been completed. All python scripts and result matrices are attached. In order to make a good comparison of the four programs about randomly selecting test cases to observe the change graph of coverage and fault detection, they will be put together below for analysis.

IV. Test of other programs

The coverage collection of the three subject programs of replace, schedule2, and totinfo is the same as the above program tcas. The following highlights the three programs that need attention. First of all, for the test of replace, we directly cd to the "source.alt/source.orig/" directory, execute "gcc -fprofile-arcs -ftest-coverage replace.c -o replace.exe", and we will find that the compilation fails (the data set is too old with grammatical standard changes). Therefore, we need to modify the "replace.c" file.

1.Insert two lines under #include $< stdio.h >$:
#include $< ctype.h >$
#include $< stdlib.h >$
2. Modify # define NULL 0 to # define NUL 0
3. Replace all getline with get_line

Now, execute "gcc -fprofile-arcs -ftest-coverage replace.c -o replace.exe" again and find only warning information, but the program can be compiled successfully. Next, you need to go to the "versions.alt/versions.orig" directory and make the same changes to the replace.c files of 32 versions from v1 to v32.

Secondly, when testing the tot_info program, it should be noted that when the command "gcc -fprofile-arcs -ftest-coverage tot_info.c -o tot_info.exe" is executed, "tot_info.c" will display errors that sin, log, and exp are not defined. Just add the definition at the LGamma function, as follows:

#define sin
#define log
#define exp

Finally, for the test of the program schedule2, the biggest difference here is that a different method is used to compare the results of the faulty version and the correct version. Since only the method of generating the script is different, the result matrix generated by the two methods should be the same as expected. First, we go to "testplans.alt" and use Sed regular expressions:

```
sed -i ' '/./{ s/^/-p[&/;s/\$/&]/}" test.universe
```

convert the universe file into one that conforms to the mts tool content format. Then remove the space between -p[and the first number, and replace the lowercase p with an uppercase P. At this time, the universe file can be converted into a "test.sh" script file using the mts tool. Run:

```
mts .. ../source/schedule2.exe ./test.universe
R runall.sh NULL NULL
```

it will generate a "runall.sh" script. Then run again:

```
mts .. ../source/schedule2.exe ./test.universe
D runalldiff.sh NULL ../newoutputs
```

to generate a compare script “runalldiff.sh”. When the “runall.sh” script is executed, move the results from outputs folder to the newoutputs folder, then copy the “versions.alt/version.orig/v1” program to source, and execute “runalldiff.sh” to directly see the different result, as shown in Fig.12.

```
>>>>>>>>running test 29
different results
>>>>>>>>running test 30
>>>>>>>>running test 31
>>>>>>>>running test 32
>>>>>>>>running test 33
different results
>>>>>>>>running test 34
```

Fig. 12. Comparative Results.

Experiments have proved that the fault detection method of this method is the same as the matrix obtained by sequentially comparing the results of tcas with the above python script. Although this method is very convenient in comparison, it is only displayed in the terminal and cannot be saved in a file, and the data needs to be recorded manually later.

V. Random Generation and Result Comparison

According to the coverage matrix generated by oneself, many test suites can be randomly selected and created. In this experiment, because there is no fixed scale when drawing the smooth line scatter diagram, all variables are self-adapted according to the input value of the abscissa and ordinate, the same is true for the random fault location map below. In addition, the smooth line in the figure is the scatter point generated based on the data first, and the data is fitted using the corresponding function. Therefore, it is reasonable for some lines to exceed 100%, and the scatter point represents the real data. The original version of the four programs and the faulty version v1 are shown in Fig.14, Fig.15, Fig.16 and Fig.17 below, so that not only can the relationship between the coverage rate and the size of the test case be compared between the correct version of a subject program and a faulty version, but also the differences and changes of the four programs can be compared up and down. The number of test cases (x-axis) for coverage is 1, 3, 5, 10, 20, 50, 100, 200, 400, 600, 800, and 1000. Specifically, first, take a test case to measure coverage, then keep the first test case, add two new test cases, and so on. Coverage (y-axis) ranges from 0 to 100%, and each graph contains two trend lines for statement (blue) and branch (red). The nodes in the figure correspond to the percentages in the table. Because the space of the graph is limited, the detailed coverage of the four programs is shown here in a table. as shown in Fig.13 The branch coverage is accompanied by the statement, and “gcov -b tcas.c” directly measures the two coverages. According to the previous experiments, because the faulty version has only one or two errors, the predicted coverage should not be significantly different from the correct version. It is predicted that the coverage rate will increase greatly at the beginning, such as 1, 3, 5, because the coverage rate starts from 0, and some use case programs will be executed randomly, but as the use cases increase, the uncovered range gradually shrinks, then you need to create a use case that has never been executed and meets the criteria, then the coverage rate will increase very slowly. To be honest, among the four programs, only schedule2 can reach 100%, and only the branch can achieve 100% in the other three programs. The guess may be that the number of test cases is not enough and the use cases given by the program itself have not reached 100%. Need Throw away the given test cases and create some new ones to test.

Coverage(%)		Test case size	1	3	5	10	20	50	100	200	400	600	800	1000
replace	original	Statement	29.02	28.43	47.45	49.41	52.16	58.04	61.18	76.08	82.75	83.92	93.73	95.29
		branch	31.11	45.56	55.56	65.56	68.89	92.22	94.44	97.78	97.78	97.78	100	100
	V1	Statement	29.39	31.76	39.61	47.45	52.16	66.27	74.90	78.04	80.00	83.14	87.06	94.44
		branch	21.11	33.33	45.56	52.22	57.78	75.56	90.00	93.33	93.33	95.56	100	100
schedule2	original	Statement	30.90	38.85	58.99	69.78	80.52	85.61	89.93	92.09	94.24	98.56	99.28	100
		branch	56.82	61.36	79.55	86.36	88.64	90.91	97.73	97.73	97.73	100	100	100
	V1	Statement	33.33	37.68	56.52	67.39	74.64	83.33	87.68	89.86	93.48	96.38	99.28	100
		branch	61.90	57.14	80.95	90.48	90.48	90.48	95.24,	95.24	97.62	97.62	97.62	100
tcas	original	Statement	46.15	76.92	78.46	80.00	81.54	83.08	86.15	87.69	89.23	89.23	98.46	98.46
		branch	15.15	57.58	72.73	66.67	75.56	72.73	78.79	96.97	100	100	100	100
	V1	Statement	46.15	76.92	78.46	80.00	81.54	83.08	86.15	87.69	89.23	89.23	98.46	98.46
		branch	15.15	60.61	72.73	66.67	75.56	72.73	78.79	96.97	100	100	100	100
totinfo	original	Statement	23.20	32.00	38.40	55.20	59.20	73.60	75.20	80.00	82.40	83.20	85.60	95.20
		branch	34.09	47.73	52.27	68.18	72.73	79.55	86.36	90.91	88.64	90.91	100	100
	V1	Statement	23.39	37.10	47.58	53.23	58.06	72.58	74.19	77.42	80.65	82.26	83.06	95.20

Fig. 13. Coverage information for the four programs.

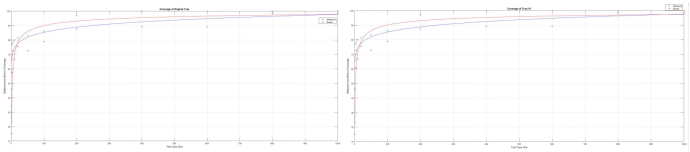

Fig. 14. The relationship between tcas coverage and test case size.

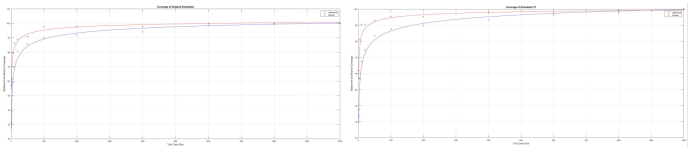

Fig. 15. The relationship between schedule2 coverage and test case size.

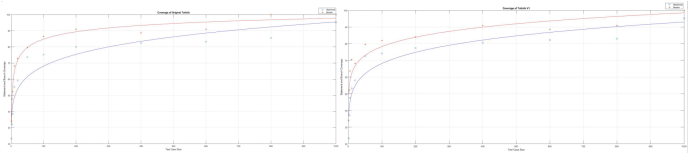

Fig. 16. The relationship between totinfo coverage and case size.

It can be seen from the figure that the coverage rate will basically increase with the increase in the number of test cases, and finally, gradually level off. When both statement and branch reach 100% or the gap between the two is small, the two lines will overlap. For example, the statement is 98.46% of tcas, branch100% and the statement and branch of schedule2 are both 100%. The coverage rate of some statements is not

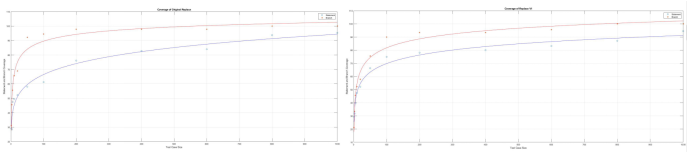

Fig. 17. The relationship between replace coverage and test case size.

100%, such as totinfo(statement:95.29%;branch:100%) and replace(statement:95.20%;branch:100%), reflecting in the graph that there will be a large gap between the two lines. In addition, it can also be observed that the branch coverage of tcas surpasses the statement as the number of test cases increases. Generally speaking, the experiment proves the conjecture that the increase in coverage rate is high in the early stage and lower in the later stage.

The experiment ended up being about how many faults were revealed as the number of test cases increased. The number of test cases randomly selected in this experiment is not the same group as the coverage rate above, because the 100 test cases used for fault location are used for the coverage rate during the experiment, it will be found that the coverage rate will reach the highest. And whether it is increased to 200, 300 to 1000 later, the coverage rate remains unchanged. Therefore, for fault detection, I use a python script to randomly take 100 test cases to locate the fault, then keep 100 and add 100 randomly to form 200 test scripts, and so on. Fig.18 below summarizes the information on how many faults are found after the execution of each group of test cases of the four programs.

Program \ case size	100	200	300	400	500	600	700	800	900	1000
tcas(v1)	7	15	25	33	42	50	54	60	69	84
schedule2(v1)	4	7	11	15	16	17	19	22	28	32
totinfo(v1)	12	24	34	41	48	58	64	71	79	85
Replace(v1)	11	17	22	25	27	33	38	42	53	59

Fig. 18. Fault detection information for the four programs.

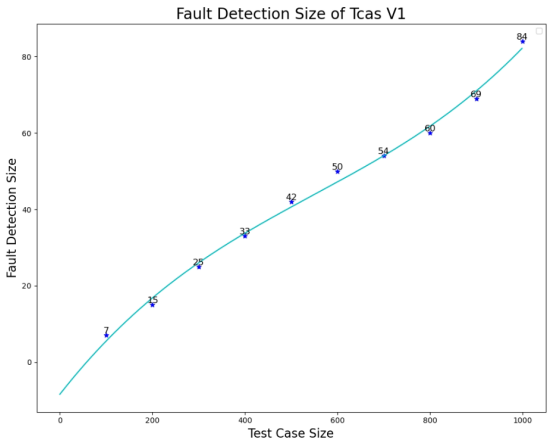

Fig. 19. The relationship between tcas fault detection size and test size.

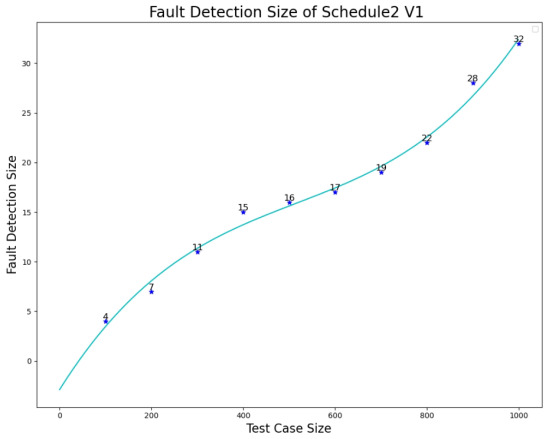

Fig. 20. The relationship between schedule2 fault detection size and test size.

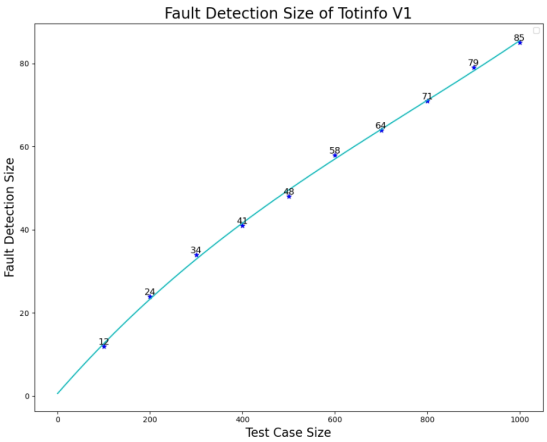

Fig. 21. The relationship between totinfo fault detection size and test size.

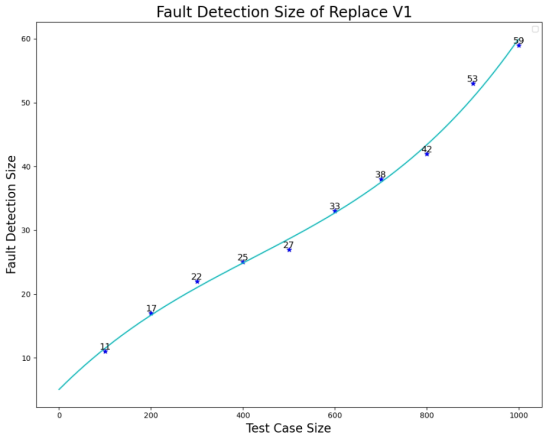

Fig. 22. The relationship between replace fault detection size and test size.

The information is reflected in each node in the graph, as shown in Fig.19, Fig.20, Fig.21 and Fig.22 above. Because the program is different, the number of detections is also different. Still, the experiment proves that as the number of test cases increases, the number of fault locations also increases. We could also see that except that totinfo has been growing linearly with the increase in the number of test cases, the other three programs start to become flat at the case size 600, and then gradually increase.

VI. CONCLUSION

Through this software testing experiment, I have mastered how to convert a program into a script, how to input the script into the program, how to use GCC, gcov to collect the coverage, and how to compare the results to determine the entire process of the fault location and improve myself the ability to solve problems. What is more, many conjectures in the experiment have also been proven to be correct through experiments. Indeed, there are also many questions during this experiment, for example: how to make the program that has not reached 100% reach 100% after the test script example is used up, and how to check and create those use cases, all of which require further in-depth testing. This course and project are very grateful to Professor Yvan Labiche for his enthusiasm and patience in answering all kinds of questions, which made the project on the right road and completed in more detail.