

ANALYSIS OF THE EFFECTIVENESS OF STUDENTS' TEST DATA

C. Stringfellow, Z. Kurunthottical
Department of Computer Science
Midwestern State University
3410 Taft Blvd
Wichita Falls, TX 76308
catherine.stringfellow@mwsu.edu, zenink@hotmail.com

ABSTRACT

One of the challenges for students is to come up with appropriate test cases to test the applications they develop. The challenge for instructors is to come up with a small test suite, yet one that is adequate to thoroughly test student's projects. This paper describes two testing techniques to aid instructors in reducing test cases, while still ensuring effective testing. The first technique uses input-output analysis and identifies relationships between inputs and outputs to generate a minimal test suite. The second technique selects certain inputs as critical points and tests in their vicinity. The main point of this paper is to compare the test suite generated by applying these techniques to a large class project to the test suite developed by students in the course using traditional approaches. While the first technique has the potential to uncover errors, results in this study show that only the latter technique uncovers errors that the students miss with traditional approaches.

INTRODUCTION

One of the challenges for students is to come up with appropriate test cases to test the applications they develop. Even a team of students in a senior or graduate level course may find it difficult. Software testing becomes even more challenging as software complexity increases, as it does in a semester team project, especially today when students work on more projects that involve visual programming environments that enable complex graphical user interactions. Students are taught the techniques for random testing, black-box testing and white-box testing, but use them with varying degrees of success. Regardless of which technique is applied, they are taught that one of their goals should be to reduce test case inputs without diminishing the efficacy of the tests.

The challenge for instructors is to come up with a test suite to thoroughly test student's projects – knowing that the students' test suites will miss some errors. Obviously, instructors want to reduce the number of test cases they use, because of time constraints. However, reducing test cases alone does not ensure effective or successful testing. This paper describes two testing techniques to aid instructors in reducing test cases. Combining the two schemes ensures that the number of test cases is kept at a manageable level, yet preserves the efficacy of the test. The first technique uses input-output analysis and identifies relationships between inputs and outputs to generate a minimal test suite [5]. The second technique selects certain inputs as critical points and

tests in their vicinity [7]. The main point of this paper is to compare the test suite generated by applying these techniques to a large software engineering class project to the test suite developed by graduate students in the course using traditional approaches they are taught.

BACKGROUND

Studies have shown that no association was found between the size of the test suite and the testing success [8]. Depending on the program size and complexity, it may not even be possible to test all the cases (i.e., due to combinatorial explosion). Black-box testing can identify important areas of the program to test based on the specifications [8]. The general perception about black-box testing is that the tester or the automated system must select numerous test cases to become as effective as a white-box test. However, no concrete evidence has been shown that white-box tests or formal methods are always more effective than black-box tests. For example, white-box testing methods are not effective when it comes to web applications because of their dynamic nature [1]. In addition, the costs of such rigorous testing can be exorbitantly high compared to black-box testing.

Menzies and Cukic [4] show that a small number of randomly selected tests are just as effective as a larger set of tests for small-scale projects. Schroeder and Korel [5] take this idea a step further, but without using randomly-selected test cases. Their study conducted an automated input-output analysis (i.e., determined relationships between inputs and outputs) that would reduce the number of tests, yet still maintain the effectiveness of the tests. Schroeder and Korel [5] mention two approaches to determining input-output relationships, namely structural analysis (requires access to the source code) and execution-oriented analysis (requires program documentation or interviews of the developers of the code). Our study analyzes the specification document. Once the relationships have been obtained, minimal test cases are derived, although it should be noted that the minimal test set might not be the optimal minimal test set [5]. It should also be noted that Schroeder and Korel's [5] study was conducted on an application that had a small set of inputs (i.e., 3 input variables and 8 input values). In contrast, our study had 76 input variables and 168 input values.

Another approach is to focus on the most likely areas that will uncover faults. Frankl et al., [2] suggest that some insight about the failure points will determine what type of testing technique needs to be implemented. The authors suggest that if the tester has foresight about the potential failures, specific testing techniques can be devised. Local exhaustive testing is such a technique that indirectly reduces the number of test cases [7]. Wood et al., [7] describe a procedure that identifies certain inputs as "critical" and focuses on all inputs close to those data values. This critical point should be something an instructor with his years of experience can identify.

An interesting point to keep in mind is that in both testing methods, the assumption is that the only errors uncovered are solely due to input. However, external failures, such as a failed library load, insufficient memory, and write-protection errors on disks, are notoriously obscure and are ignored in this study [6].

METHOD

This study is applied to a project completed by a team of five students in a graduate software engineering class. (Unfortunately, the other team of students did not complete their project in time.) The system functions similar to an application for a car rental agency. Activities include making and canceling reservations, maintaining service reports, keeping track of available cars, etc. Visual Basic 6.0 and Microsoft Access were used for developing the system. Graphical user interfaces (GUIs) were primarily used for user inputs and for outputs. This section describes the testing approach to a software application project, modeled after two separate techniques by Schroeder and Korel [5] and by Wood et al. [7]. It compares this test suite to the students' test suite.

GUI input text-boxes or dialog windows represent input variables of the problem domain [5]. Input values are the actual data that are entered into a text-box or dialog window for an input variable. Input values can also be entered by selecting a radio button or a check box.

The basic idea behind this approach is to go through every interface beginning from the login screen and main menu to the logout screen to determine all paths. Since this is not an error-testing suite, all input data generated must be valid functional values that would not result in an error based on the specifications.

A large software application may have many input variables. Hence, for the first step, only two data values are chosen for each input variable. The second step vectorizes (or combines) related information as one input variable. For instance, let us say an application requires the name and address of a person. Instead of counting the name and address separately as two input variables, the name and address are combined as one input (string). This significantly reduces the number of input variables to test. Some of the input variables may contain dependencies (i.e., password is dependent on username). Some of the dependent data would be invalid (i.e., correct username but wrong password or vice-versa) and is eliminated, thus only a test suite with valid input is obtained. The last step determines critical points. These critical points are areas where more errors are prone to occur, and ensures that any errors overlooked in the input-output analysis are found. The instructor is responsible for deciding which points serve as critical inputs. Table 1 summarizes the steps taken:

Table 1. Steps in obtaining minimal number of test cases.

<u>Steps</u>	<u>Action</u>
1.	Create a minimum of two data values for each input variable.
2.	Combine (or vectorize) related input variables (ex. Customer Name, address,etc.).
3.	Sum up the number of total input variables and input values.
4.	Calculate the total number of input value combinations (feasible and infeasible combinations).
5.	Eliminate the infeasible paths to obtain a reduced test suite.
6.	Derive a minimal test suite by retaining key test cases and purging redundant or recurring input.
7.	Determine critical points (such as input points that would be expected of producing more errors) and conduct testing around those data values.

RESULTS AND ANALYSIS

The MAVIS Rent-A-Car software and its interfaces are briefly described herein. The initial interface is the login screen where either the manager or the employee has to login with the username and the associated password. Once the correct data has been entered in the login interface, the main menu options are shown. Figure 1 illustrates the main menu interface that the *employees* can access. The only difference between the manager and the employees is that the *manager* can also access the last four options (i.e., print reports, update car inventory, update rental rates and update employees). From this interface any one of the 13 choices (including “LOGOUT”) can be made, 12 of which take the user to the subsequent corresponding screens.

Figure 2 shows an interface that contains several input variables that are closely related. Not vectorizing inputs results in 13 input variables (the RAF number and customer id are already given and need not be entered). With vectorization, this interface has only 3 input variables. Vectorizing certain inputs does not compromise testing efficiency in these instances, because, the data entered are just supplementary information that has no bearing or effect on the overall testing process.



Figure 1. Main menu options.

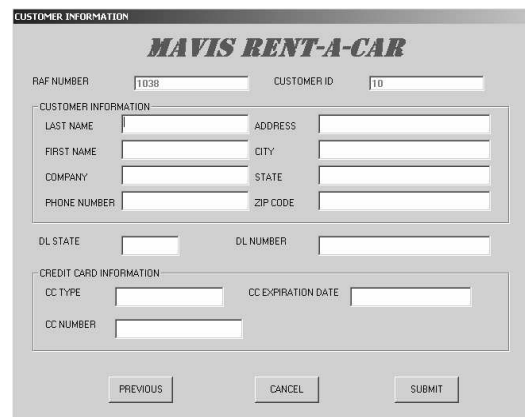


Figure 2. Customer reservation form.

Input-output analysis identifies 76 input variables and 168 input values. (A tree structure can be used to show all the combinations.) The combination of all paths, feasible and infeasible, results in a very large number (2.1516×10^{25}) of test cases. Fortunately, a significant number of these are infeasible. Only three main test subsets are created (i.e., for the manager, service representative and an agent). These test subsets include 9258 possible combinations (5202 (manager) + 2028 (service representative) + 2028 (agent)), already a significant reduction in test cases. Another more arduous approach is to compute the infeasible paths, then compute the feasible paths (feasible = total – infeasible combinations). Thus, we know that the infeasible number is very close to 2.1516×10^{25} . Note that the manager’s test subset has more combinations since there are more main menu options for the manager.

A test set of 9258 test cases is still quite huge, so a minimal set is generated. The technique identifies the first interface with the largest number of input variables. This is the main menu screen, which contains 13 input variables. Since this is the stage at which all other interfaces branch out, this is used as the defining part of the minimal set. The tree structure is pruned, so that branches executing the same function are cut.

The minimal set for tasks both managers and employees can perform consists of 14 input paths. Tests were designed where one employee was assigned to perform about half of the tasks, while the other employee performed all but one of the remaining tasks. Finally, cases to test the ability of the manager to perform manager tasks and the one remaining employee task were designed. This involved 12 input pathways. The assignment of a test path to the type of employee eliminates repetition of the same test path with another employee or manager. This procedure resulted in 26 test cases.

The minimal test suite did not uncover any errors in the case study system. The reason for this could be that each test data point has input values that are valid and the students' application handles valid input correctly. (This was an exceptional group of students.) Finding the absolute minimal set is an NP-complete problem [5]. Therefore, even though this study arrived at a minimal set, it remains to be seen if there is an even better optimal minimal set.

While the combination of these techniques has the potential to uncover errors, results in this study show that only the latter technique uncovers errors that the students miss with traditional approaches. The second method of local exhaustive testing does catch errors not discovered by the input-output analysis technique and not discovered by the students' test suite. The course instructor determined that the critical points were those inputs where dependencies occurred (i.e., one input is dependent on another and interchanging these data values would generate errors). These critical points are username-password and date-time data. In this testing technique, both valid and invalid test cases were applied. Note that if the application takes appropriate measures to catch or circumvent inappropriate input, then an error does not exist. A test is successful only if it catches an error not handled by the application. For example, in the application, the username and password must match in order for a login to be successful. The students' application caught all errors involving mismatched username and passwords (i.e., invalid entries). In other words, either a wrong username or password prevented the user from progressing to the next screen. Nine test cases were designed using the critical point test technique for username-password. Testing did not discover any errors for these input variables.

Date and time for a reservation were supposed to be checked when the "check availability" and "make reservation" tasks are performed. Fourteen test cases were designed and eight of the tests were successful. The errors discovered involved time conflicts (for example, the time a car was returned occurred before the time a car was picked up when the date of rental and date of return were the same).

The test suite designed by the graduate students using traditional approaches uncovered a total of 14 errors, but most of these were invalid entry errors (i.e., where the user entered invalid data and the software did not catch it). However, their test cases did find errors that this study did not discover. For instance, one of the errors found in the system allowed the user to pick up a car not reserved for that date. Another error the students' test suite found was that the daily rate remained the same even after applying the "update daily rate" option. Other errors the software engineering team discovered included a wrong calculation of the final rental amount and the "cancel" button remaining inactive in the service report interface. Using the combined input-output analysis and critical point test technique, the errors found by the test cases were primarily due to the dependencies mentioned earlier.

CONCLUSION

This paper compares the test suite generated by applying input-output analysis and local exhaustive testing techniques to a large class project to the test suite developed by students in the course using traditional approaches. Using input-output analysis and local exhaustive testing should enhance the effectiveness of the testing process (i.e., uncover more errors). Another aim of the input-output analysis method is to enhance the efficiency of the testing process by reducing a huge number of test cases to a minimal set. The larger test suite designed by the graduate students using traditional approaches did find errors that the input-output analysis method did not discover, but most of these were invalid entry errors.

Local exhaustive testing is used to ensure that errors overlooked by input-output analysis may be caught. This method did catch errors not discovered by the students' test suite. In addition, if invalid data and additional critical points were implemented, new errors might have been uncovered.

REFERENCES

1. Elbaum, S., Karre, S., and Rothermel, G., Improving web application testing with user session data. *In Proceedings of the 25 IEEE International Conference on Software Engineering*, (May 2003, Portland, OR), 49-59.
2. Frankl, P., Choosing a testing method to deliver reliability. *International Conference on Software Engineering*, (1997), 68-78.
3. Kaner, C., Falk, J., and Nguyen, H.Q., *Testing Computer Software*, International Thomson Computer Press, U.S.A, 2000.
4. Menzies, T., and Cukic, B., When to test less. *IEEE Software*, (Sep.-Oct. 2000), 107-112.
5. Schroeder, P.J., and Korel, B., Black-box test reduction using input-output analysis. *In Proceedings of the 2000 ACM Special Interest Group on Software Engineering International Symposium*, 2000, pp.173-177.
6. Whittaker, J.A. and Thompson, H.H., Black-box debugging. *Queue*, (Dec.-Jan., 2003-2004), 68-74.
7. Wood, T., Miller, K., and Noonan, R.E., Local exhaustive testing: A software reliability Tool. *In Proceedings of the 30th Annual Southeast Regional Conference*, (April 1992), 77-84.
8. Yu, Y.T., Ng, S.P., Poon, P., and Chen, T.Y., On the use of the classification-tree method by beginning software testers. *In Proceedings of the 2003 ACM Symposium on Applied Computing*, (2003), pp. 1123-1127.