Experimental Methods in Computer Science

[Software testing - Bug detection using test suites]

Jani Hilliaho, Sebastian Rehfeldt

ABSTRACT

Software companies put a lot of effort in testing their programs to reduce the probability of having bugs in their software. During this work, it is analyzed how many of all realistic bugs, which would still exist in the code, could be detected by test suites. Therefore a list of common JavaScript bugs was created and concrete examples of them were injected in three Node.js programs. Afterwards, different test suites were tested to determine their capabilities of finding the injected bugs. In the end, confidence intervals for the maximum bug detection probability are presented and it is discussed how the probability depends on characteristics of the test suites and programs.

1. INTRODUCTION

Software bugs which lead to undesired behaviour of programs can cause big costs and troubles for companies. To minimize the amount of software bugs, companies develop their software according to quality metrics and test the functionality of their software exhaustively. Whereas software metrics can already give advices which modules are complex and hence, a possible source for errors, test suites can give even more hints about the presence of bugs, up to the exact location or type of the bug. But therefore testing needs to be done right.

During this report, it is discussed how the probability of detecting bugs can be estimated by the corresponding test suites and how it changes with the line test coverage and with the amount of test cases. Therefore, bugs are injected into different programs and different test suites are run to estimate the proportion of bugs found. Given the results of the test runs, confidence intervals for the probability of finding bugs are calculated to answer the research question.

Research question:
What is the probability of a test suite to detect the software faults (i.e., the bugs) that still exist in the code of a program?

The experiments are conducted on three JavaScript (node.js) programs with multiple test suites each. For these programs confidence intervals for the maximum bug probability are presented and it is analyzed how the bug probability depends on the test coverage/ amount of test cases and if it depends on the complexity of programs.

The remainder of the report is structured as follows: The next section gives an introduction of the used programs and discusses the independent variables. Also typical bug types of JavaScript are described in section 2. The experimental results are presented and discussed in section 3. Finally, the report will be rounded off by a conclusion of our results in section 4.

2. EXPERIMENTAL SETUP

This section introduces the different programs, bug types and describes the setup of the experiments.

To run the experiments, three different node modules which are publicly available on Github were used. Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine.[1] Node.js runs on server-side and is frequently used in programming web servers of all kinds. The big advantage of Node.js for this work is that there are hundreds of thousands small to big modules publicly available.[2] The installation and testing of packages are easily done on every operating system using the node package manager, npm.[3]

Before describing the programs in more detail, it can be helpful and important to understand which independent variables are considered in the experiment and which typical bugs occur in Node.js programs.

2.1 Introduction of the dependent and independent variables

The dependent variable in this experiment is clearly defined as the probability of a test suite to detect bugs and is calculated as fraction of bugs detected and bugs injected. Therefore, for each program and test suite a binary vector is created which indicates whether an injected bug was found by the test suite or not. For the results it is not necessary to count how many test cases or test suites failed as software engineers are already alarmed as soon as just one test case fails.

For the independent variables two categories are considered: description of programs and description of test suites.

Clearly the bug probability depends on the characteristics of a program. To analyze how they influence the bug probability, the **complexity of programs** is considered as one independent variable. Obviously, complex and long programs are more likely to contain bugs. With the differentiation into complex and simple programs, based on cyclomatic complexity, it is possible to examine if a change in the complexity leads to a significant change of the bug probability.

[1] https://nodejs.org/en/
[2] http://www.modulecounts.com/
[3] https://www.npmjs.com/

As all of the programs are run in the Node.js runtime, there is no need to consider the operating system or browser as an independent variable.

One obvious choice for an independent variable describing the test suites, and probably the mostly considered metric in the reality, is the **test coverage**. The test coverage, here line coverage, gives evidence on how many lines of code are covered by the test suites. It is desirable to have a high test coverage as bugs which are injected in uncovered lines cannot be detected by the test suites. To analyze how the bug probability increases with the test coverage, multiple test suites with high (80-100%), medium (60-80%) and low (<60%) test coverages were created. In practice a 100% test coverage is associated with the detection of all possible bugs in the program, but this is obviously not true. To prove and analyze this, a further level to this variable is added which describes whether the test suite consists of many test cases or if the test coverage was achieved with a minimum amount of test cases. In the experiments it was already possible to reach a 100% test coverage with a very small number of test cases and this test suite cannot be as good as the maximum test suite.
Another independent variable describing the test suite is the **number of test cases**. This variable is used to visualize how the test coverage and probability of detecting bugs increases during the creation of a test suite. To create reliable plots, very fine-grained levels, depending on how many test cases are available, are considered for the programs. For the programs, test suites consisting of e.g. 1,2,3,5,10,20,... test cases were created and tested. These results could also be grouped into high, medium and low number of test cases. One possible way would be to compute the ratio of test cases used and total test cases and take the first third as low, the second as medium and the last as high.
Furthermore, the **type of testing** can also be considered as an independent variable. In general, there are two types of testing, White-Box and Black-Box testing. Whether the first one takes advantage of knowing the code and enables to directly test specific functionality, Black-Box testing only tests whether the program gives the correct output for a specific input. It could be interesting to know which approach leads to a higher bug probability.

There are more variables possible in the experiment, but we decided to keep them fixed. In that way, only JavaScript programs are used and only one bug is injected at a time. By keeping these variables fixed, the complexity of the experiments is kept small and enables for better conclusions.

2.2 Bug types

For the experiments, it was necessary to create a list of common JavaScript bugs as there is no such a list available to our best knowledge. The created list is based on personal experience and different articles about mistakes in JavaScript.[4][5] Furthermore, we tried to adopt some of the bugs from the bug list (C programs) provided by our teacher. To do that, each C bug was checked whether this bug is also plausible in JavaScript. Obviously that is not the case for every bug type as JavaScript does not use pointers and differs greatly in many language features. Nevertheless, some bugs of the list were added to our bug list.
In total the bug list contains 12 different bug types which, to our best knowledge, try to cover most of the typical JavaScript bugs. The bugs are presented and discussed in the following. All bugs types allow the code to run without crashing, but can lead to a different behaviour and wrong results.

1. **Missing curly braces:** Curly braces are not mandatory after if, else, for or while statements. Omitting them is not considered as a syntactical bug and the code is running without crashing. Whether the code runs fine and as expected when only one statement is executed inside these loops/conditions, omitting braces leads to bugs when multiple statements are executed in the loops/conditions as only the first statement will be considered to belong inside the loop/condition.

   In practice JavaScript developers tend to omit braces when only one statement is needed inside a loop or condition. The real bugs occur when the code needs to be changed later and more statements need to be added. In many cases the programmers forget to add the braces and wonder later where their bugs come from.

2. **Automatic semicolon insertion:** JavaScript does not force to use semicolons at a line-ending. These semicolons are inserted automatically during parsing. Nevertheless, missing semicolons can lead to problems and result in bugs which are very hard to detect.[6] For example when a return statement is split over multiple lines, a semicolon will automatically be inserted in the line of the return which cuts-off the rest of the return statement which is written in the line after.

   In practice, many JavaScript developers are not aware of this feature and simply ignore semicolons. That is why this bug type was added to the bug list.

3. **Not understanding type coercion:** In JavaScript it is not necessary to define the type of variable during declaration as it is a dynamically typed language. Whether this should make the coding easier, it can lead to bugs when comparing variables of different types. For example the character 5 would be equal to the number 5 when using two equal signs as a comparator as this comparator automatically converts one type to the other. To prevent this automatic conversion, JavaScript developers use three equal signs instead of two. Another possible source for errors of that type are switch statements which implicitly use the three equal signs.

   As many JavaScript beginners are not aware or just ignore this problem and this could result in very nasty bugs, this bug is part of the bug list.

4. **Confusing addition and concatenation:** The + operator is overloaded in JavaScript and used by addition and concatenation. Problems can occur when the developer is not taking care of the types of the used variables and e.g. tries to add a character 5 to a number 5 which would give 55 as a result of concatenation.

[4] http://tutorialzine.com/2014/04/10-mistakes-javascript-beginners-make/
[5] http://www.w3schools.com/js/js_mistakes.asp
[6] http://cjihrig.com/blog/the-dangers-of-javascripts-automatic-semicolon-insertion/

5. **Arithmetic operations with floats:** Like in many other languages, working with float is not very precise and leads to unexpected errors, e.g. 0.1+0.2 = 0.30..04. This is not always considered by the developers and can lead to incorrect results or failing comparisons.

6. **Use assignment instead of comparison operator:** This is a typical bug in many languages and often occurs in while loops or if statements. As while and if statements are frequently used in the code and it could easily happen to forget an equal sign, this bug type is likely to appear in the real world without leading to crashes in the code but wrong results.

7. **Misplacing semicolons:** Additionally to missing semicolons, misplaced semicolons can also lead to bugs, e.g. Java or C developers which are forced to add semicolons at line-endings could tend to add a semicolon after an if or for statement and before the code block. This would prevent the code block from running and would lead to serious bugs.

8. **Undefined is not null:** In JavaScript null is used for objects and undefined is used for variables. If an object is not defined, it is undefined and only when it is defined it can be null. This is often not understood and used incorrectly by inexperienced developers.

9. **Missing break in switch-case block:** Like in other languages, e.g. in Java, a missing break will also result in the execution of the following cases which is often not intended.

10. **Forgetting var:** JavaScript uses the var keyword to declare variables inside a scope. When the var keyword is missing, the variable will be declared globally which can lead to bugs which are almost impossible to detect. For example a loop could use a counter which is declared globally. Inside the loop, a second function is called using the same variable name and replacing the loop counter by some other variable resulting in an incorrect execution of the loop.

11. **True or false in logical expressions:** When using logical expressions in an if-else statement, it sometimes can happen that one variable is always true or false which would result in the execution of one and the same branch only. This bug type was adopted from the bug list provided by our professor as this bug is also considered as reasonable in JavaScript.

12. **Wrong logical operator:** Especially in loops the number of iterations is often bounded by a less-or-equal or smaller-than operator. Being not careful with these operators could lead to a missing or to an additional pass of the loop and create a wrong result.

With the defined bug list we try to cover as many JavaScript bugs as possible. We therefore used types of really common bugs from other languages and JavaScript specific bugs, like the ones concerning type coercion, semicolon insertion etc. Only with bug types which achieve a big coverage of possible bugs, general conclusions about the bug detection probability can be derived. Also it is necessary to inject the bugs in a realistic proportion to get correct results for all JavaScript programs. Unfortunately, no such a list or distribution is known to us and is probably out-of-scope for this report to create a realistic bug type distribution over all possible JavaScript programs. Also not every bug type from our list leads to wrong results which can be detected by the bug list. So a missing var will mainly lead to the same result as with the var, except that one variable is globally declared. More research needs to be done to examine how severe these bug types are to create a better bug list with realistic proportions of bugs.
Nevertheless, the above list should enable to draw conclusions on the bug probability with a, to our best knowledge, general validity. The following section introduces the programs and explains in the example of one program how the bug types are instantiated and injected into the programs.

2.3 Programs

During this work three programs of different domains are used. It is not possible to derive conclusions with full generality with them as it is only an educational scenario, but still it is a simplified real world experiment with practical importance. Nevertheless, we tried to find programs of different domains and ended up with a command-line calendar[7], a formatting module for the command line[8] and a vector library for physics.[9]

2.3.1 Calendar.js

The purpose of this program is to create a basic calendar in the command line and to return the day for a given date. It uses basically only one file which is of medium size. Obviously, shorter programs are more likely to contain bugs and have a lower complexity. To keep track of this, the cyclomatic complexity of the program was examined using plato[10]. The lines of code were measured by 78 and the derived cyclomatic complexity was 18. Another important measurement for the experiments is the test coverage. In the case of the calendar, the line test coverage is 100% and it was achieved using white-box testing. With this approach every function was invoked with some input and compared against the expected output. The coverage was assessed using blanket.js.[11]

The calendar should also serve as an example what bugs were injected into the programs and how different test suites were created.
We decided to search for if, for or while statements in the code and then removed the braces around the code block. We used this two times as instantiation of the first bug type of our list. For the second bug type we searched for return statements and separated the statements over multiple lines. In total this bug type was used twice in the code. A probably more common bug is the usage of the assignment operator instead of the comparison operator. We used three examples of this bug by replacing the comparator by the assignment operator. For the 8^{th} bug we replaced one occurrence of undefined by null. Also we added semicolons on two spots after a for or if statement to instantiate the misplacing-semicolons bug type. The next 3 bugs are representative for bug type 12

[7]https://github.com/ramalho/calendar.js
[8]https://github.com/timoxley/columnify
[9]https://github.com/dead-horse/pvector/
[10]https://github.com/es-analysis/plato
[11]http://blanketjs.org/

and change the amount of iterations inside a loop by changing a less-or-equal to less-than or vice-versa. Also we added a true or false to if statements to represent bug type 11.
The last bugs are very hard to detect and not necessarily lead to wrong results. Anyway they are bugs and frequently occur in JavaScript codes. We therefore removed var keywords from equally named variables as one example for bug type 10 and removed one equal sign from the three equal signs three times.
In the end, we ended up with a list of 20 bugs which covers 9 out of 12 bug types and is presented in table 1.

Type	Amount	Description
1	2	Braces were deleted
2	2	Return statements were splitted
3	3	"===" replaced by "=="
4	0	No possible spots available for addition/ concatenation in Calendar
5	0	No possible spots available for float bug type in Calendar
6	3	Replace "==" by "="
7	2	Semicolon added after if/ for
8	1	Replace undefined by null
9	0	No switch statements in Calendar
10	1	Var before equally named variables were removed
11	3	True or false were added in conditions
12	3	"=" replaced by "<=" in loop and vice-versa

Table 1: Bug list for Calendar.js

Whether this probably does not represent perfectly all JavaScript programs and the real proportion of bugs, we tried our best to create an experimental setup from which realistic conclusions can be derived which are true for most of the JavaScript programs and bugs. A similar list with an equal proportion is used for the other two programs to keep the results comparable.

To run experiments with different levels of test coverage, additional test suites for the programs needed to be created. Therefore test suites from the full test suite were removed to lower the line test coverage. In the case of the calendar, the full test suite contained 35 test cases in 9 test suites. In the end, 6 levels of coverage: 100%, 90%, 75%, 60%, 45% and 30% were achieved. In addition to that, minimal versions of these suites were created by removing test cases which do not lower the test coverage. By doing so, it was possible to analyze whether the test coverage alone is sufficient to predict the bug probability.
Also test suites with increasing number of test cases were developed to analyze the curve of the bug probability by a given amount of test cases. Starting with a test suite consisting of just one random test case, randomly selected test cases were added to the test suite until the full test suite was reached.
In the last preparation step, permutations of the full test suite were built to compute the confidence interval of the maximum bug probability. This was done by randomly removing test cases from the full test suite. In the case of the calendar, the test suite permutations consisted of 30 out of 35 test cases.

2.3.2 Columnify

The purpose of this program is to make almost any console output easier to read. The program can be used for reformatting data to be shown as columns. The program consists of files width.js (6 lines), utils.js (193 lines) and index.js (297 lines), where index.js includes the main functions, utils.js includes some functions used in index.js, and width.js loads one module needed for running the program. All bugs were injected to Index.js because it includes the main functionality of the program. Input, output and usage examples for the program can be found at GitHub[12].
Columnify has 25 test suites including total of 139 test cases. The line coverage of all tests is 99.54%, and cyclomatic complexity is 46. The complexity was calculated using plato, and the coverage with istanbul[13].
The test suites test the functionality of columnify with some inputs, comparing the outputs to expected outputs, which means black-box testing. As the types and amounts of injected bugs are the same than in Calendar.js, they can be found from table 1. As the result of tests, 7 of 20 bugs were not detected at all. All tests run with Calendar.js were also run with Columnify.

2.3.3 PVector

This program is meant for vector calculations. The program consists of just one file, pvector.js, having 178 lines. Input, output and usage examples for the program can be found at GitHub[14]. Pvector has 16 test suites including a total of 28 test cases. The line coverage of all tests is 95.1%, and the cyclomatic complexity is 30. The complexity was calculated using plato, and the coverage with istanbul.
The test suites tested every function of the program with some inputs, comparing the outputs to expected outputs. The types and amounts of injected bugs were copied from Calendar.js if possible, but some changes were made because some bug types used in Calendar.js could not be injected to Pvector. The used bugs in Pvector are shown in table 2.

Type	Amount	Description
1	2	Braces were deleted
2	4	Return statements were splitted
3	2	"===" replaced by "=="
4	0	No possible spots available for addition/ concatenation in Pvector
5	0	No possible spots available for float bug type in Pvector
6	2	Replace "==" by "="
7	3	Semicolon added after if/ for
8	0	No possible spots available, replace undefined by null
9	0	No switch statements in Pvector
10	1	Var before equally named variables were removed
11	3	True or false were added in conditions
12	3	"=" replaced by "<=" in loop and vice-versa

Table 2: Bug list for Pvector

[12]https://github.com/timoxley/columnify
[13]https://github.com/gotwarlost/istanbul
[14]https://github.com/dead-horse/pvector

All tests run with Calendar.js and columnify were also run with Pvector except the tests about minimum and maximum test suites with certain test coverage because it wasn't possible to achieve some test coverages with any combination of test cases.

2.4 Injection of bugs

This section describes how bugs were injected and which results were collected from the test suites.
In case of the Calendar.js, bugs were added manually to the program. Afterwards, all test suites were tested whether they could detect the bug. By using qunit as a test framework it was really fast and easy to run the tests. The results were collected in a table and were given by binary data indicating whether a bug was found or not.
To make every test easily repeatable and to allow permutation tests with high amount of permutations, some scripts were developed for injecting bugs and running the tests with Columnify and Pvector. At first, both programs had own files for each single test suite. To make the used test suites and test cases easier to select, all test suites were combined to create big test scripts for both programs. The test scripts were also equipped with a feature which parses testing parameters from used program arguments. At the end of the work, the test scripts were run with arguments selecting a bug to inject and the used test cases. At first the tests were run with the mocha[15] framework, but the istanbul framework was also used for collecting coverage data during every test.
Most of the tests with these two programs were run by running the test commands directly in the terminal. Because thousands of single tests with different sets of test suites were run during the permutation tests, a feature of selecting used test cases randomly was added to the test scripts. Multiple bash scripts were also created for running test scripts hundreds of times from command line and for saving the results in a form which could be easily used in R.
The resulting data of each program was imported to R and then analyzed by plots and statistical tests. The results are presented and discussed in the following section.

3. RESULTS AND DISCUSSION

This section introduces the experimental results and analyzes them using plots and hypothesis testing. In the end, the results are discussed regarding the research question.

3.1 Confidence intervals for bug probabilities

As the research questions asks how many bugs can be detected by a test suite, confidence intervals are calculated for the bug probability for a given program and test suite. The confidence interval is estimated for each program in separate based on permutations of the maximum test suite. Permutations of the full test suite are used to get more results from which a distribution of the maximum bug probability can be derived. A single run would not be statistically sufficient for describing the bug probability. Before the confidence intervals are calculated, the process of collecting data is described in more detail.
Starting with the maximum test suite, permutations of it were created by randomly removing a fixed amount of test cases. Then single bugs were injected into the programs and all test suites were run to check whether the bugs were detected or not. This was done for every of the 20 possible bugs of a program. The bug probabilities were calculated from the binary vectors of each test suite and used to form the distribution of the maximum bug probability from which the confidence interval was calculated.

3.1.1 Confidence interval of Calendar.js

5 out the 35 test cases of the full test suite were removed randomly to create the permutations. In total, 10 test suites for the calendar were created by this procedure. The derived probability vector is the following: (0.75 0.75 0.65 0.75 0.60 0.75 0.60 0.70 0.75 0.75). It is easy to see that most of the test suite found 75% of all bugs and that this is the maximum. As a consequence of this, the bug probabilities are not normally distributed which can also be proved using the q-q plot in figure 1 and with the shapiro-wilk test.

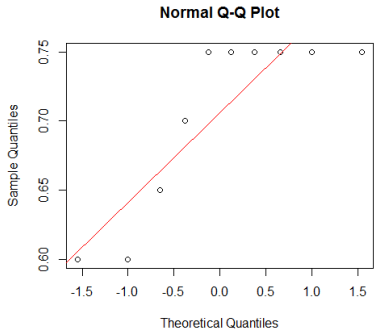

Figure 1: Q-Q Plot for the bug probabilities of Calendar.js

In normally distributed data, the line in the q-q plot would be a diagonal which is here not really the case. Also the shapiro-wilk normality test declares the data as not normal with a p-value of 0.001269. As a consequence Bootstrap is used to finally obtain a statistically reliable confidence interval. In this approach 1000 random samples of the probability vector are created with replacement. For each of the samples the mean is calculated and collected in a new vector. This new and sampled data can be analyzed in figure 2.

[15]https://github.com/mochajs/mocha

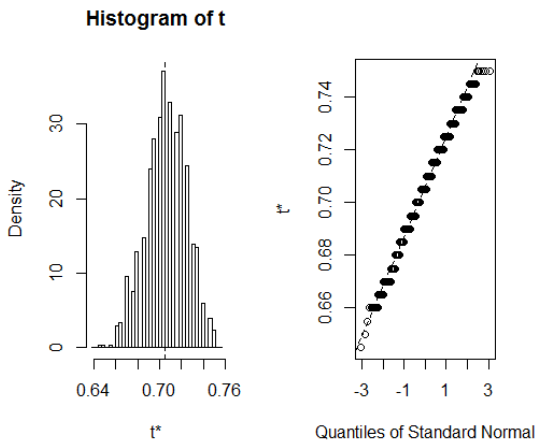

Figure 2: Bootstrapped data for Calendar.js

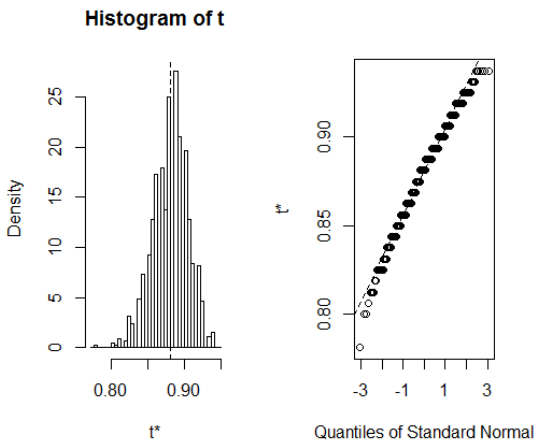

Figure 3: Bootstrapped data for the bug probabilities of Calendar.js with 16 bugs

The bootstrapped data now clearly is normally distributed and confidence intervals can be computed based on this data. The 0.95 confidence interval calculated with the adjusted bootstrap percentile (BCa) method is 0.65 to 0.735. So it can be said that the real probability of detecting bugs with the maximum test suite lies in that range with 95% confidence. Probably the real bug probability is a bit higher as some test cases were removed to create the permutations but it should not make a big difference.

One more insight from the calendar was that the last 4 injected bugs were never detected. That proves the assumption that a missing var or a double equal for comparison are hard or impossible to detect by test suites. If these bug types cannot be detected by the test suites and are better covered by a linter like JSLint[16], these bugs were pruned out from the results as they lower the bug probability. In a second run, still not normally distributed without the sampling, bootstrap gave the following results which can be seen in figure 3.

This shows normally distributed data and that the confidence interval is now higher than before. The real bug probability lies now in the range of 0.81 to 0.92. The created confidence interval shows that a test suite with many test cases and 100% line test coverage detects the majority of bugs but is lower than 100%. Whether finding most of the bugs is considered to be really helpful in practice, it might be interesting why not every bug was detected. We think that this is a really optimistic goal which is hard to reach, because bugs can be so varied and special that it is hard to test against everything. In the case of the calendar, the test suite is maybe also not big enough and test cases might be missing to test every control flow of the program.

3.1.2 *Confidence interval of Columnify*

Nearly the same permutation test as for Calendar.js was run with Columnify. As permutations, 20 out of 25 test suites were randomly selected and a total of 100 test suites were programmatically created for Columnify. The minimum of derived probabilities was 0.36, average 0.44 and maximum 0.6. When removing the last 4 bugs not detected in any program, the minimum was 0.44, the average was 0.55 average and 0.75 was the maximum probability of detection. The bug probability wasn't normally distributed which is also proved using the q-q plot in figure 4 and with the shapiro-wilk test.

[16]http://www.jslint.com/

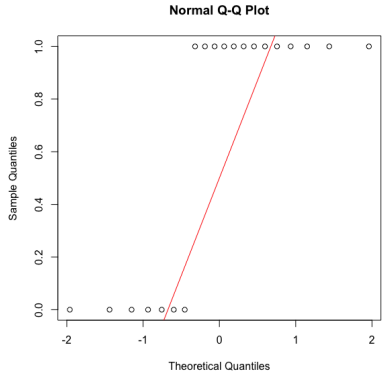

Figure 4: Q-Q Plot for the bug probabilities of Columnify

As stated with Calendar.js, the resulting data is not normally distributed because the q-q plot is not diagonal. The shapiro-wilk test also declares the data as not normal with a p-value of 0.000003538. Finally, the same bootstrapping method as with Calendar.js was used to derive a statistically more reliable confidence interval. This new and sampled data can be analyzed in figure 5.

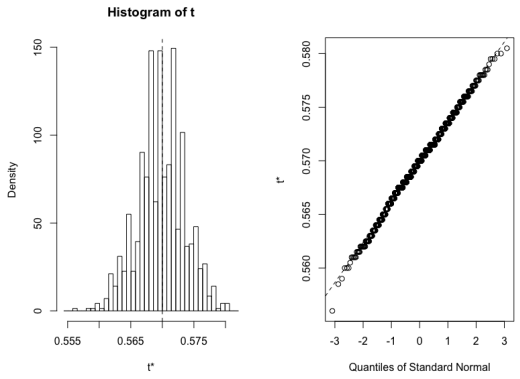

Figure 5: Bootstrapped data for the bug probabilities of Columnify with 20 bugs

The bootstrapped data is now clearly normally distributed and confidence intervals can be computed based on this data. The 0.95 confidence interval calculated with the adjusted bootstrap percentile (BCa) method is 0.5620 to 0.5765. In a second run, the last 4 bugs were pruned out as with Calendar.js The bootstrap gave then the following results which can be seen in figure 6.
This shows normally distributed data and that the confidence interval is higher now. The real bug probability lies in the range of 0.703 to 0.721 with high probability. The interval is much slimmer than in the Calendar due to the higher amount of permutations.

3.1.3 *Confidence interval of Pvector*

The same permutation test than with Calendar.js and Columnify was also run with Pvector. As permutations, 12 of 16 test suites were randomly selected and a total of 100 test suites were programmatically created for Pvector. The minimum of the derived probabilities was 0.35, average 0.54 and

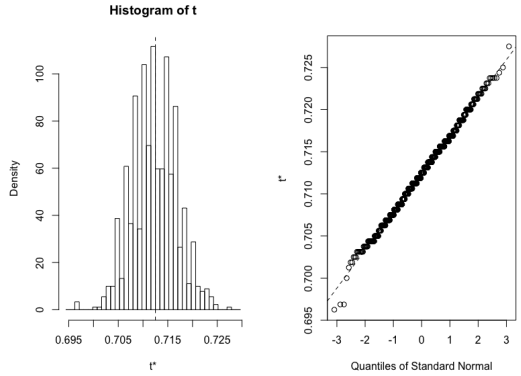

Figure 6: Bootstrapped data for the bug probabilities of Columnify with 16 bugs

maximum 0.7. Because it wasn't possible to inject exactly the same bugs for every program, Pvector had only 3 of 4 bugs not detected in any programs. When we pruned out these 3 bugs, we got 0.41 as minimum, 0.64 as average and 0.82 as maximum probability of detection. The bug probabilities are not normally distributed which is proved using the q-q plot in figure 7 and with the shapiro-wilk test.

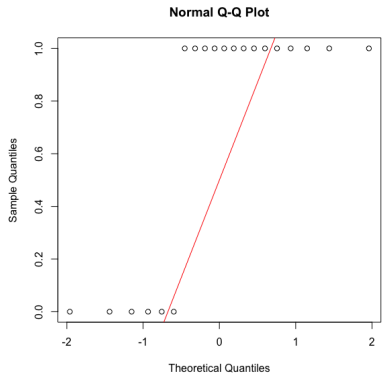

Figure 7: Q-Q Plot for the bug probabilities of Pvector

As in previous sections, the data is not normally distributed because the q-q plot is not diagonal and the shapiro-wilk test calculates a p-value of 0.000001857. To get a statistically more reliable confidence interval, the same bootstrapping method as described before is used. The results can be analyzed in figure 8.
The 0.95 confidence interval calculated with the adjusted bootstrap percentile (BCa) method is 0.528 to 0.557. As with other programs, the bugs not detected in any programs were pruned out from the results as they lower the bug probability. In a second run, bootstrap gave the following results which can be seen in figure 9.

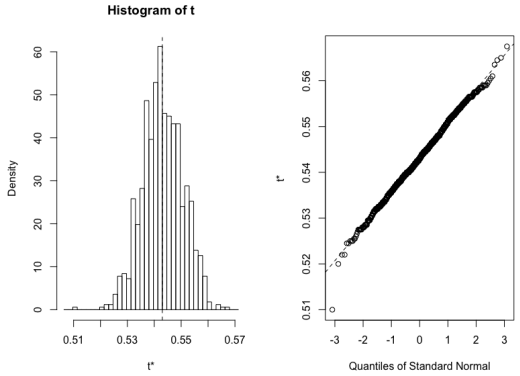

Figure 8: Bootstrapped data for the bug probabilities of Pvector with 20 bugs

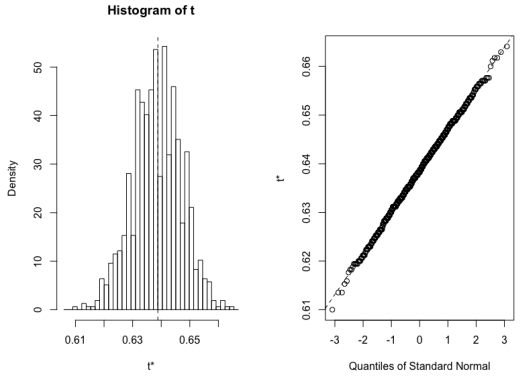

Figure 9: Bootstrapped data for the bug probabilities of Pvector with 17 bugs

Now, the real bug probability lies with high probability in the range of 0.6215 to 0.6554 which is around 10% higher than before.

3.2 Bug probability by test coverage

Reasonable intervals for the maximum bug detection probability are given with the confidence intervals. For the development of test suites it is interesting how the bug probability increases with the test coverage or test cases. The curve can also give evidence when the probability of detecting bugs saturates and stops increasing. When time is limited, the development of the test suites could be stopped then.
To analyze the impact of the test coverage, experiments on different coverage levels were conducted for all the programs and all available bugs. The bug probability for each level and program is then visualized in a plot to show the slope of the bug probability. Figures 10, 11 and 12 show the plots for all three programs.

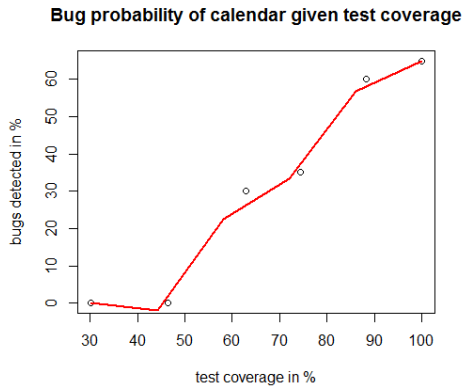

Figure 10: Bug probability on test coverage - Calendar

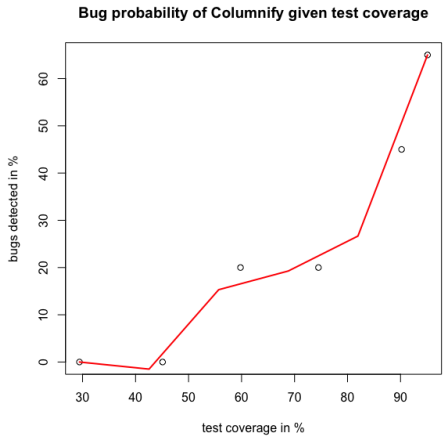

Figure 11: Bug probability on test coverage - Columnify

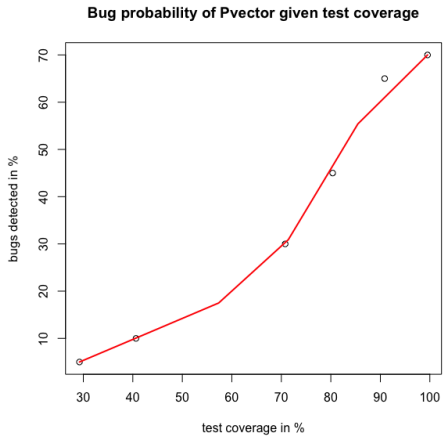

Figure 12: Bug probability on test coverage - Pvector

The plots show that the first bugs were detected after coverages of around 40-60% were reached, which is surprisingly late. We assume that the reason for that is that the bugs are

injected in code parts which are not covered by the tests by coincidence and bad luck. Afterwards the probabilities are strongly increasing until high coverages are reached. When the test coverage is already high, the bug probability is believed to increase slower than before which can hardly be seen in the plots of Calendar and Pvector, and not at all in the plot of Columnify. Also we would like to mention that the probability is quite low in the end. This is the case because we have not pruned out the last three or four bugs which were never detected. Also it can be seen that the bug probabilities of some programs are lying above the confidence intervals from the former section. This can be explained by the need of removing test cases for the calculation of the confidence intervals and underlines that the real confidence intervals are a bit higher than calculated in our experiments. Similar plots without these bugs are possible but the curve will probably be the same. It is also worth mentioning that the curves are decreasing in the beginning. This is not possible in practice when higher coverage levels contain all test suites of lower levels. The curves decrease because we applied a smoothed line to the data points.
The curves are also strongly dependent on the particular test suites and the order in which they were added. With bigger programs and more runs, better and more reliable curves could be created.

3.3 Bug probability by test cases

In a similar analysis it could be studied how the amount of test cases affects the bug probability and the test coverage. To draw plots and analyze the results tests for all programs were conducted on the test suites based on test cases. The slope of the bug probability for the calendar is shown in figure 13 and the plot for the increase of test coverage by adding test cases is given in figure 14.

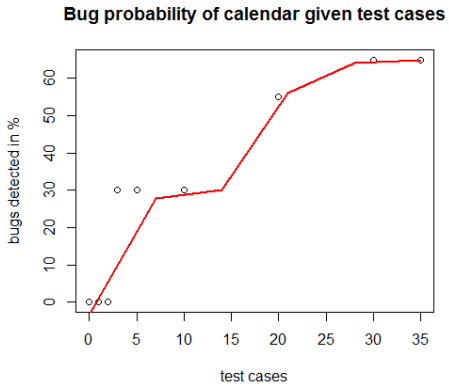

Figure 13: Bug probability in relation to the amount of test cases - Calendar

The plots show that the bug probability and test coverage are almost increasing in the same way. By adding the first test cases both percentages increase strongly. When reaching around 20 test cases, the rise becomes weaker. For the test coverage this is easily explained by reaching the 100% coverage which is the maximum. The reason for the stagnation of the bug probability is similar to the one given in former experiments.
The shape of the curve is highly dependent on the test cases

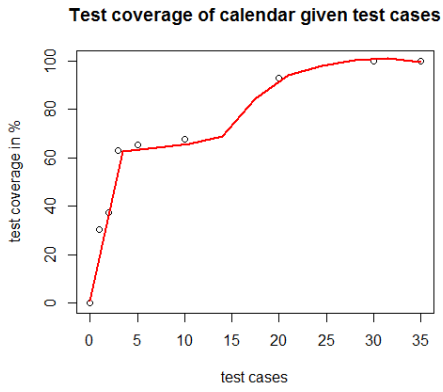

Figure 14: Test coverage in relation to the amount of test cases - Calendar

which are added first. So no general conclusions can be drawn. The curves for the other programs are presented in the following and it is checked if they match the observations from the calendar. A reliable way of creating the curves would be to run multiple experiments where the order of test cases changes, but we consider this to be out-of-scope of this project to conduct experiments on that and refer to the idea of running multiple experiments if a reliable curve is really necessary.

The slope of the bug probability for the Columnify is shown in figure 15 and the plot for the increase of test coverage by adding test cases is given in figure 16.

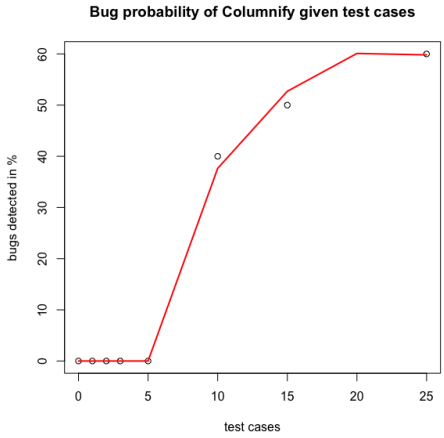

Figure 15: Bug probability in relation to the amount of test cases - Columnify

The bug detection curve shows that the bug probability is zero with less than 5 used test cases, but increases rapidly after it. The reason for that is probably that the bugs are injected in code parts which are not properly covered by test suites. The test coverage curve shows that the coverage increases fast before 5 test cases and slowly after it. This shows that an increase in test coverage does not necessarily lead to a better bug detection probability.

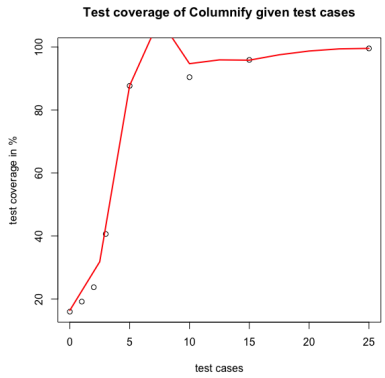

Figure 16: Test coverage in relation to the amount of test cases - Columnify

As stated with Calendar.js, the shapes of the curves are highly dependent on the order of the test cases, which prevents making general conclusions.
The slope of the bug probability for the Pvector is shown in figure 17 and the plot for the increase of test coverage by adding test cases is given in figure 18.

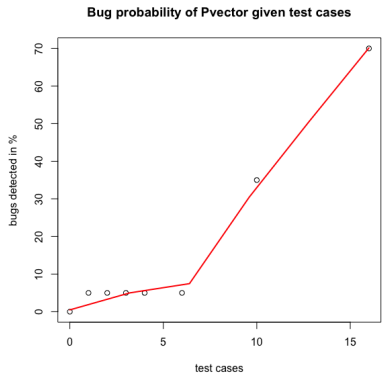

Figure 17: Bug probability in relation to the amount of test cases - Pvector

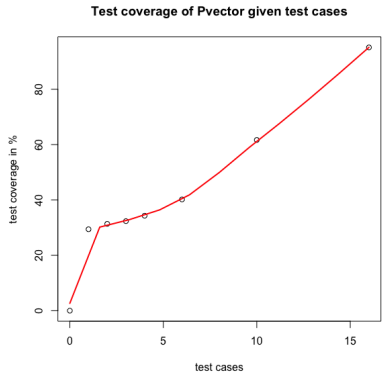

Figure 18: Test coverage in relation to the amount of test cases - Pvector

The bug detection curve shows that some bug is detected with just one test case. After it, the curve stays in the same value until 6 test suites is reached and increasing rapidly after it. The test coverage starts from about 30% with just one test case, and rises smoothly after it. Also here the bug probability is lower than the test test coverage.
The conclusions from the curves of all three programs are that bug detection probabilities and test coverages increase as the number of test cases increases. In some cases, a small amount of test cases is not enough for detecting any bugs. The reason for that is probably that the bugs are injected in code parts which are not properly covered by test suites. As stated before, the shapes of the curves are highly dependent on the order of the test cases, which prevents making more precise conclusions about the curve shapes.
From a practical point of view it could be interesting if software engineers should head forward high amounts of test cases or a high test coverages and which metrics gives more evidence on the bug probability. Due to time limitations, we just want to give an idea how this could be analyzed in an experiment. At first levels for high coverage and high amount of test cases need to be set. Therefore 80-100% could be a high test coverage. Similarly, 80-100% of all available test cases from the maximum test suite could be considered as a high amount. More levels are plausible by using the same approach. The experiments would results in multiple bug probabilities for each approach which form a distribution. Afterwards it could be checked with help of statistical tests which distribution contains the higher probabilities.

3.4 Comparison of minimal and full test suites

It was already shown that the bug probability increases with the test coverage. In the programs a high coverage could already be achieved by a few test cases which obviously do not form a reliable test suite. This section presents results for full test suites and minimal test suites which have the same coverage level but with significantly lower amounts of test cases. A test suite with more test cases should lead to a better bug probability. The following hypothesis was tested to verify this intuition:

$$H_0 : P(bugs \mid full\ ts) \leq P(bugs \mid minimal\ test\ suite)\ . \tag{1}$$

To give an intuition about the data and the problem, a plots are given in figures 19 and 20.

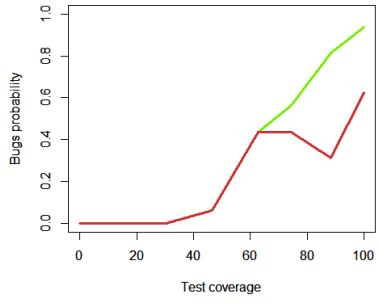

Figure 19: Bug probability for full (green) and minimal (red) test suites in Calendar.js

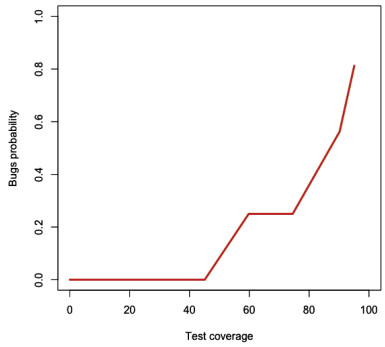

Figure 20: Bug probability for full (green) and minimal (red) test suites in Columnify

The plots show that the full test suite is at least as high as the minimal. This is always true as it contains all test cases from the minimal test suites. The plot shows the difference between minimal and full test suites clearly with Calendar.js, but only one curve can be seen in the plot of Columnify, because the bug detection probabilities are always same with the minimal and full test suites. The interesting point is, that the curve of Calendar.js starts to differ at the level of 75%. This shows that it makes no big difference in the beginning whether many test cases are used or just a minimal amount. This can be explained by the low probabilities which cannot differ very much. The differences get bigger with a higher coverage. An interesting measurement is the low probability at 90%. This can be explained by the creation of test suites. We probably removed a test suite which covers code where many bugs were injected. This test suite is then later added back to the other test suites to reach the coverage levels. The creation of the test suites is therefore not optimal, but it was due to the fact that the calendar program is relatively short and test cases can already cover big parts of the code.
Given the q-q-plots in figures 21 and 22 which do not show diagonals and the small amount of data available, the results are not considered to be normal and sufficient for an analysis.

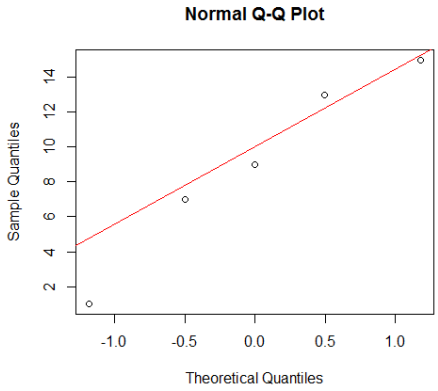

Figure 21: Q-q-plot for the full test suites of Calendar.js

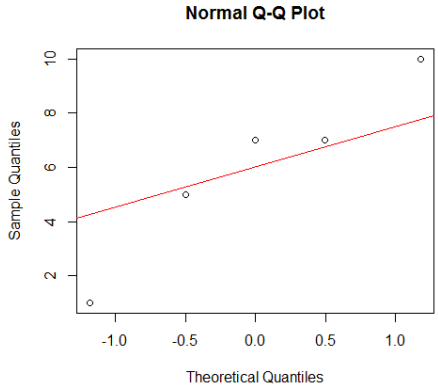

Figure 22: Q-q-plot for the minimal test suites of Calendar.js

To test the hypothesis, a new data vector was created which represents the full data minus the minimal data. Finally bootstrap was applied on the new vector to calculate the confidence intervals. The results can be seen in figure 23.

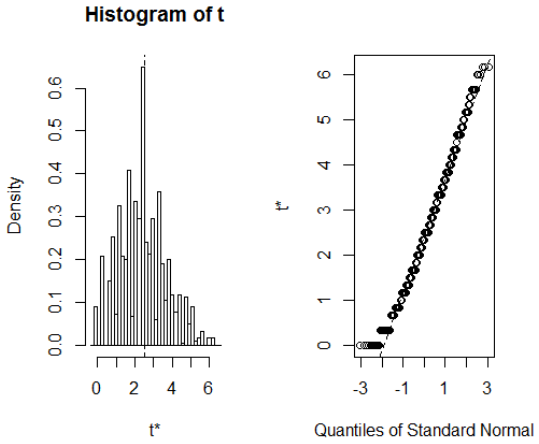

Figure 23: Bootstrap for the full data vector minus minimal data vector of Calendar.js

The bootstrap data shows that the confidence interval is above 0 and the real difference lies in the range of 0.3 to 5.2 with high probability. It can be concluded that the full test suites detect more bugs than minimal test suites, in average around 3 out of 20 bugs were detected by the full but not by the minimal test suites.
A different and maybe more reliable way to examine the quality of the test suites would have been to create permutations for the full maximal and full minimal test suites and run the experiments on them. This excludes the factor of test coverage from the experiments and leads to only one change in the setup. Also the difference would have possibly been higher as the maximal and minimal test suites mainly differ when high coverages are achieved.
As a consequence for the real world, it can be concluded that test suites with high line coverages are not necessarily able to detect the majority of bugs in the software and that it is crucial to the quality of test suites to have test cases for all

possible inputs and control flows.
Because the curve of minimal test suite is same as the curve of full test suite with Columnify, the difference and confidence interval of them are always zero.

3.5 Bug probability by complexity

As stated in previous sections, the bug detection probability increases with the test coverage and the size of the test suites.
To detect any bug, there need to be a test suite that covers the code line where the bug is injected. However, some code lines can be used in many branches with different values of variables and it is possible for some bugs to have an influence on the program only with some values. Therefore it is assumed that a test suite with a high branch coverage detects bugs more likely than a test suite with lower branch coverage.
However, as the line coverage is used to measure test coverages during this work, it is possible that all branches of a complex program are not always tested. That means that the complexity of the used programs can have an influence on bug detection probability. The following hypothesis was tested to verify this intuition:

$$H_0 : P(bugs \mid low\ complexity) \leq P(bugs \mid high\ complexity) \quad (2)$$

To compare the bug probabilities with programs with different cyclomatic complexities, the values of Calendar.js and Columnify are shown in table 3.

	Calendar.js	Columnify
Test coverage	100%	99.54%
Amount of test cases	35	139
Cyclomatic complexity	18	46
Confidence interval, all bugs	0.81 - 0.92	0.703 - 0.721
Confidence interval, 16 bugs	0.65 - 0.735	0.562 - 0.577

Table 3: Cyclomatic complexities and bug detection confidence intervals of Calendar.js and Columnify

As shown in the table, the bug detection probability of Calendar.js is much lower than with Columnify, and Calendar.js also has a lower cyclomatic complexity. It is not possible to draw reliable general conclusions from these values because there is so much other differences between the programs. As stated in the previous sections, a higher amount of test cases and higher test coverage usually means better results in bug detection. The test coverages are nearly same in both of these programs and Columnify has much more test cases than Calendar.js. Because Calendar.js has still a higher probability of bug detection, it can be assumed that cyclomatic complexity has an effect on the probability of bug detection and that at least with these programs the hypothesis is true.

3.6 Bug probability by testing approach

When deciding on the testing approach, it could be helpful to know whether White-Box or Black-Box testing is superior. In the beginning, we mentioned the testing approach as an independent variable and suggested to analyze its' impact on the bug probability. We therefore wanted to test programs which were tested with different approaches. Unfortunately, this would not allow good conclusions as differences could also be explained by the usage of different programs of varying complexities.
To analyze which approach is superior, experiments on one and the same program needed to be conducted. Due to a lack of time we are not able to create second test suites using the other approach for one of the programs and cannot test this hypothesis.
Nevertheless, the experiment would be similar to the former ones. Two test suites would be needed for one and the same program. Afterwards a probability distribution could be derived and tested using the usual statistical tests.

3.7 Discussion

In this last section, the initial research question is discussed based on the results of our experiments. Unfortunately, it is not easy to directly answer the question as the probability depends on different variables like the complexity, test coverage, amount of test cases etc. Nevertheless, it can be concluded that the maximal probabilities in the programs lie in the ranges of 81-92%, 70-72% and 62-66% when certain bug types were pruned out. Also it can be stated that the bug probability rises with the test coverage and the amount of test cases until it stagnates at a maximal level which is probably below 100%. No really reliable results can be drawn on the shapes of the curves as they are highly dependent on the order of added test cases. We only assume that the rises of the curves are higher in the beginning and are getting weaker with high amounts of test cases or high test coverages. One more important conclusion from our work is that the differences between minimal and maximal test suites are small in the beginning of the development of a test suite and only grow when higher test coverages are reached. Based on the comparison of programs with different complexities, programs with lower complexity seem to have higher probability of bug detection.
We want to end up this part with mentioning implications for the real world. Software tests are a great tool to detect the majority of bugs in software, but they cannot guarantee that all bugs are covered and they need to be done right. So the line test coverage is frequently overrated and misunderstood. This means that a 100% line test coverage alone does not mean that the test suites are perfect. Many test cases are needed to cover the big variety of possibles bugs.
Software tests alone may also not be enough to create high quality software which is almost free of bugs. In addition to the tests, linters and code checkers could be used to detect bug types which are hardly detectable by tests and rarely lead to real errors. Finally, software metrics could be used to detect modules with high complexities which could be then simplified by splitting into multiple modules which are less probable of containing bugs. The best software quality can be achieved by using a combination of all these approaches. This helps companies to develop software with almost no bugs and prevents from high costs due to errors.

4. CONCLUSION

During this report, we introduced common bug types for JavaScript programs. Based on a bug list with 12 different types, bugs were instantiated and injected into three different programs. We performed multiple experiments on the programs and analyzed how the probability of a test suite to detect bugs depends on the amount of test cases, the test

coverage or the complexity of a program. Furthermore, we calculated confidence intervals for the maximum bug detection probabilities of our test programs. The intervals are calculated after pruning out some bug types which better should be detected using linters and the highest values lie in ranges of 81-92%, 70-72% and 62-66%. This shows that test suites, if they were created correctly, can detect the big majority of bugs, but that they probably cannot detect all possible bugs. Another conclusion is that some bug types were never detected by our test suites. For these bugs, code checker and linters are a more probable choice to detect them and prevent them from causing errors.
In our first assignment of this course, we showed how to predict the occurrence of bugs in programs based on software metrics. With this work, we now studied a second tool for developing high quality software and keeping errors low. Additionally to the occurrence of bugs, software testing can also directly give hints on the location and type of bugs which is more powerful than using software metrics only.

4.1 Future Work

As we parsed the bug list from many bug lists in the Internet, we do not have a reliable list of bug types and their distribution which is representative for all JavaScript programs. A more reliable list would allow to draw more general conclusions on the bug probability.
More programs could also be used for the experiments to verify whether our conclusions are still valid for them.
A more advanced and elaborate approach could also try to find a formula which calculates the bug probability given the values for the independent variables.
The programs and scripts used in this work can be found from GitHub[17].

5. CONTRIBUTIONS

Sebastian

- writing abstract + introduction + experimental setup (excluding description of pvector and columnify and their testing)
- writing result section (excluding results of pvector and columnify and the comparison of simple and complex programs)
- writing conclusion and future work
- assessment of program complexities and test coverage of calendar
- creation of test suites for calendar and running the experiments
- preparing the R scripts

Jani

- Selecting 12 bugs and creating the bug list
- Writing description of Pvector and Columnify
- Creating testing scripts, finding bugs and running tests for Columnify and Pvector
- Comparing experiment results of Columnify and Pvector to the results from Calendar.js
- Writing "Comparing bug probabilities of simple and complex programs"
- Proofreading the report

[17]https://github.com/jhilliaho/EMCS-design-of-experiments