Verification and Validation of an Open Source Product

*Part-I: Automated Static Analysis*

Ratna Pranathi Garigapati

930108-9042

<raga15@student.bth.se>

Chilla Kartheek Arun

930416-3778

[kach15@student.bth.se](mailto:kach15@student.bth.se)

Medisetti Meghana

930103-P168

[meme15@student.bth.se](mailto:meme15@student.bth.se)

Srikar Reddy Nadikattu

930923-1554

[srna15@student.bth.se](mailto:srna15@student.bth.se)

Sharen Polavarapu

940521-2573

[sapo15@student.bth.se](mailto:sapo15@student.bth.se)

Nitin Parasa

940803-0238

[nipa15@student.bth.se](%20nipa15@student.bth.se%0c)

***Abstract* — This paper presents the data of comprehensive static and dynamic code analysis of an open source “logic-based”, “number-placement” puzzle Sudoku. For the user to enjoy the puzzle it is most important to test it extensively. As a part of this test, the automated static code analysis has been done with the help of the tools Findbugs and PMD. We have selected these tools basing on the type of errors they detect and that help in increasing the efficiency of the testing process.**

# **INTRODUCTION**

Our report has two sections: Part I deals with the automated static code analysis and the Part II discusses about the dynamic code analysis.

Over the years many approaches have been developed for program analysis. Automated Software analysis involves the use of automated tools to determine the attributes and behavioral patterns of the code at different phases. Program analysis can be broadly categorized into two types namely, Static analysis and Dynamic analysis. Static automated analysis involves analyzing the structure of the code at the time of compilation to explore the possibility of any code irregularities at later stages.

As part of this assignment, we were assigned the task of performing automated static analysis for the source program of Sudokuki, a Java based Sudoku puzzle game. Sudoku is a well-known puzzle game that was developed in 1979. The main strategy of this single-player game is to fill the 9x9 grid that consists of nine blocks where each block has 3x3 cells such that each of the rows, columns or 3x3 cells contain distinct digits 1-9. The developer randomly fills some cells of the grid with these digits [1]. The unique solution to the game depends on the player’s logic.

1. **INSPECTION GOALS**

Before commencing the code analysis phase of this project, we laid out a list of preliminary inspection goals. These inspection goals represent common code vulnerabilities that are identified by most automated static code analysis tools [2].

1. Control Flow Defects
2. Input-Output Defects
3. Class and Method Defects
4. Data Flow Defects
5. Comparison or Relational Defects
6. Declaration Defects
7. Computation or Numeric Defects
8. Data Reference Defects
9. Exception Managements Defects

After performing code inspection on the given source code, we shortlisted our inspection goals based on commonly encountered warnings. Eventually, after discussion among team members, we arrived at the following inspection goals. Our objectives are to check for defects such as:

1. Control Flow Defects: It includes common code vulnerabilities such as loop terminations, nesting, exception handling and checking for all possible switch-case statements.
2. Declaration Defects: It pertains to all variable, attribute and constant declarations apart from typecasting and initialization of attributes and variables. It also examines the access modifiers.
3. Data Referencing Defects: It involves checking if values are within the declared bounds and object or array reference are non-null.
4. Concurrency Defects: It includes common defects including race conditions, occurrence of deadlocks, and non-availability of resources. Synchronization on Boolean data.
5. Comparison Defects: It checks for anomalies in conditional statements, comparing a string using operators verifying the correctness of Boolean expressions.
6. Computation/Numeric Defects: It checks for common code vulnerabilities such as computational overflows, operator precedence and parentheses checking.

The above mentioned goals are generalized categories of recurring coding defects found in most Java programs. We verified the defect log to observe warning patterns encountered.

The inspection goals chosen have had a great influence on the process for selecting a suitable automated static code analysis tool. The goals provide the entry criteria for selection of static analysis tools. This has been documented in the next section.

1. **TOOL SELECTION**

Static code analysis tools aim to highlight the potential threats that prevail in the program that an organization develops.

This process is done by thoroughly scanning the source code and finally the bugs are given as the output for a single run of this tool on the source program. The objective of this assignment is the verification and validation of an open source software product, hence this involves the usage of certain tools. Initially we performed an extensive search for various static code analysis tools that would be helpful for us to find the bugs that are prevailing in the software product given to be tested. The software given to us is written in Java and this lead to a search for tools for finding bugs for Java. With the help of the Internet, we could find a list of 9 most commonly used static code analysis tools. There were several other tools available such as Coverity and Klockwork which guaranteed accurate results, but they could not be employed as they were commercial applications requiring a fee. Analyzing the benefits and limitations of each tool extensively was not possible as we had very limited timespan and knowledge. Considering the inspection goals mentioned above, the set of tools have been categorized either as “Accept” or “Reject” for them being used to carry out a test to inspect the source program for errors. Table 1 consists of the above information. Each tool has been keenly inspected to find out that tool which satisfies all the inspection goals we mentioned here.

Carefully examining the available tools, the kind of bugs they can detect, the features we came to a conclusion that both Findbugs and PMD are the appropriate tools that well address the inspection goals.

Initially while getting to know about the tools listed in the Table 1, some tools such as Bandera, JLint, IntelliJ IDEA and SonarQube were rejected as they do not produce outputs that address our inspection goals and rather focus on the code quality and language manipulation in Java. ThreadSafe could be used but it is built to mainly detect the concurrency bugs but not the other prevailing defects in Software Testing. Considering SonarQube and IntelliJ IDEA, they required some prerequisite knowledge for operating. IntelliJ IDEA is an Integrated Development Environment (IDE) rather than a plugin in some other IDE’s such as NETBEANS and ECLIPSE. Hence IntelliJ IDEA is not considered for the analysis process. SonarQube is yet another tool which mainly focuses on gathering code quality metrics but not analyzing the source code of an application to detect for errors. Since, we didn’t have prior knowledge about how to use it, these options had been ruled out. Another set of tools are ESC/Java and Checkstyle. Both these tool have their own styles of analyzing the code of the application, where ESC/Java works well by creating annotations and CheckStyle by code conventions. Due to lack of idea regarding these tools, they were rejected too.

Finally we were left with two tools that closely produce outputs covering all our inspection goals. The tools are Findbugs and PMD where each tool has its own importance and the kind of bugs they detect. Both the tools that we have selected for static code analysis do not cover all the inspections goals, however most of the goals are addressed with the help of Findbugs and some addressed using PMD. We noticed that results produced by PMD were given in more detailed manner when compared to Findbugs. In Table 2, we have depicted the inspection goals achieved by tools Findbugs and PMD can detect when run on the given source code.

Findbugs [3] is an open source static code analysis tool for the Java based programs that has been developed by University of Maryland. It checks the syntax of the source program with common erroneous coding practices to detect errors [4]. PMD [5] is an open source static code analysis tool for Java based projects to detect possible bugs like “empty try-catch blocks, dead code and classes with high cyclomatic complexity measurements, unnecessary objects and so forth. However, both PMD and Findbugs have their own setbacks. E.g. they fail to detect bugs such as “divide by zero, array length less than zero and unreachable code” [6].

The motivation for favoring certain tools over others has been described in detail in tabular form in Table 1. Further, we have taken the help of relevant literature to support our choice of tool selection.

1. **CODE REVIEW METHOD**

We have selected “Software Walkthrough” to conduct static code analysis. It is a common technique used for peer code reviewing. It is considered to be an informal way of code inspection. Walkthrough method is a standard IEEE specification, consisting of three inspection roles: - Walkthrough Leader, Reader, and Recorder. We have altered these roles to cater to our requirements but their overall essence is preserved [7].

**Walkthrough Leader** (moderator): an inspector who presides over code inspection process. The person in this role performs administrative tasks such as scheduling meetings, shortlisting automated tools, finalizing inspection goals, and concluding outcomes for individual inspection items.

**Reader**: an inspector who examines each inspection item individually and discusses the relation between the warnings generated and possible inspection goals involved with other team members.

**Recorder**: an inspector who is tasked with recording all the relevant proceedings of code inspection meetings. The recorder keeps track of team member’s opinions on generated warnings, alternative approaches, warning count, and inspection time.

Our inspection team was split up into three groups with two members, with each group taking up one of the above mentioned roles.

1. **CODE REVIEW PROCESS**

Code review was performed using the following procedure:

* Planning: In this phase, all roles were designated for each inspection team member. Further, inspection goals and frequency of inspection meetings were decided after holding a discussion.
* Individual Inspection: Each team member observed the warnings generated and documented whether they considered them to be true positives/false positives.
* Meeting: The readers provided a briefing for all discussion items. Each team member reflected upon their views on warnings. The defect data is collected and a consensus was reached for each warning based on mutual agreement. Meeting proceedings and log time were noted by the recorder.
* Rework/Follow-Up: All members made amendments to their defect description reports after briefing conducted at the meetings. The recorder makes final entries into the defect summary report.

After planning a system for inspection, each warning was inspected individually by all team members. Each team member had their own perception about each warning, classifying them as true positive, false positive and uncertain. All opinions on warnings were discussed at the scheduled meetings and final decision was taken for placing the warnings in one of the three classes.

The warning count in the defect summary report does not match the total number of errors generated by the tools - PMD and Findbugs. This is because all the redundant warnings were merged into single warning descriptions.

1. **ANALYSIS OF RESULTS**

The results were systematically organized based on the review process. The warnings generated were divided into three categories:

1. True Positive: A condition where a warning is generated pointing out coding errors in the test result when there actually exists a defect.
2. False Positive: A condition where a warning is incorrectly generated indicating coding errors for correct code.
3. Uncertain: This case is observed when the member of the review team are unsure about the warning(s) generated. This is due to inexperience of the testers and a lack of understanding of complex coding concepts.

The most common defects found by automated static analysis tool Findbugs and PMD on testing the given source program can be generalized as:

Dodgy code, code correctness, malicious code vulnerabilities, multi-threaded correctness, irregular try-catch and finally blocks.

We referred to the bug descriptions provided in the official site of Findbugs to analyze the defect summary report for each individual warning as some of the warnings generated involved complex Java concepts [8]. This was helpful in classifying the warnings into different classes: true positive, false positive and uncertain.

## **CONCLUSION**

After analysing the results, we observed certain aspects of automated static code analysis. They have been discussed below:

**Scalability and Efficiency**: This refers to the ability of the automated static analysis tool to be able to analyse large chunks of code in lesser amount of time, while generating the least possible number of false positives in the process. In terms of number of warnings generated and time taken by the tool, it is generally observed that PMD generates significantly higher number of warnings compared to Findbugs [9]. Considered n case of Sudokuki, we observed 132 warnings for Findbugs and 235 warnings for PMD. In case of time taken, it took 25.59 seconds for PMD warnings and 2.67 seconds.

**Overlapping Defect Categories**: After analysing the results of static analysis, we came across a variety of defect types covering different inspection goals. We found both overlapping as well as disjoint defect types for the warnings from PMD and Findbugs. The defects that were commonly found in both were empty try-catch blocks and empty if-statements. Apart from these, all other warnings generated were different from each other. According to [9], overlapping defect categories include Null dereferencing errors, concurrency errors and array out of bounds errors.

**Integration with popular IDEs**: Since most of the code analysis is done simultaneously with development, it has become increasingly important for tool developers to create plugins for the easy integration of automated code analysis tools with the best IDEs in the market [9]. The main idea is that IDEs provide better interfaces and usability. Although, most of the automated tool developers have their own APIs, they also create plugins for different versions of IDEs. Most of the automated tools that we surveyed had plugins for IDEs like Maven and Ant. However, not all of them had plugins for popular ones like Eclipse and NetBeans. For instance, JLint and Bandera are software APIs and do not provide integration through IDE plugins.

Table 1: Selecting the Static code analyser

|  |  |  |  |
| --- | --- | --- | --- |
| S. No | Static Analysis Tool | Decision | Motivation for Accept/ Reject |
| 1 | FindBugs [site] | Accept | * Based on “Bug pattern” [9][4] * Compares syntax of the source code with common erroneous practices to detect the errors [4] * Its uses “simple intraprocedural data flow analysis” [4] * Allows users to write custom bug detectors in Java [4] * It is easier to use as fewer and more precise warnings are generated [4] |
| 2 | IntelliJ Idea | Accept | * It is more of an Integrated Development Environment(IDE) rather than being a static code analysis tool * Prior knowledge and deep understanding is needed to implement it |
| 3 | ESC/JAVA | Reject | * It is based on “theorem proving” and checks the attributes of source code [4] * It follows a different approach when compared to other tools and requires external annotation assistance as it more it is more effective when used with the annotations [4] |
| 4 | PMD [site] | Accept | * Conducts “syntactic checks on program source code” * It also provides users to choose the detectors or group of detectors to be run. * It provides a more detailed description of the errors * Allows users to write new bug pattern detectors in Java * It invokes the JVM for each and every iteration which lowers its speed and lowers its speed by 20% and affects the performance [4] |
| 5 | JLint | Reject | * Builds a lock graph and checks for cycles in the graph to detect the deadlocks, however it is known to report the same warning multiple times on the same line * “Jlint is not easily expandable” * Generates too many warnings thus increasing the probability of false positives * Jlint produces many false positives in case of null dereferencing, array bounds warnings. * It has compatibility issues with latest Java versions [4] |
| 6 | SonarQube | Reject | * It is mostly focused on gathering code quality metrics [10] |
| 7 | ThreadSafe | Reject | * It is designed to mostly detect the concurrency defects * It is not an open source software [11] |
| 8 | Bandera | Reject | * It does not take into account the standard Java library calls and just limits the “usability and applicability of Bandera” [4] * It is specialized in detecting the deadlocks while other potential bug patterns are neglected * Since it cannot analyse the Java library it is only suitable for small code modules [4] |
| 9 | CheckStyle | Reject | * It deals with checking conformance with coding conventions * It is known to generate false positives while testing the “code readability issues” [12] |

Table 2: Comparing FINDBUGS and PMD:

|  |  |  |
| --- | --- | --- |
| INSPECTION GOALS | FINDBUGS | PMD |
| Control Flow Defects | Yes | No |
| Declaration Defects | Yes | Yes |
| Data Referencing Defects | No | No |
| Concurrency Defects | Yes | No |
| Comparison Defects | Yes | Yes |
| Computation or Numeric Defects | No | No |

Table 3: Time log of the reviewers:

|  |  |  |  |
| --- | --- | --- | --- |
| REVIEWER | VIOLATIONS EXAMINED | | TIME TAKEN (minutes) |
| FINDBUGS | PMD |
| Nitin Parasa | 132 | 235 | 220 |
| Ratna Pranathi Garigapati | 132 | 235 | 210 |
| Medisetti Meghana | 132 | 235 | 210 |
| Sharen Polavarapu | 132 | 235 | 200 |
| Srikar Reddy Nadikattu | 132 | 235 | 190 |
| Chilla Kartheek | 132 | 235 | 200 |

Part II: Dynamic Analysis

***Abstract*— This section of the report deals with the dynamic code analysis of the open source software Sudokuki. Following the rules as per the IEEE 829-2008 format the document has been prepared. The focus here is to analyze the test case generation with the help of the tools that have been selected. The annotation of comparison between the dynamic code analyses performed by the selected dynamic code analyzers is provided.**

1. **INTRODUCTION**

As discussed earlier in the introduction to automated static analysis, there exists another type of analysis that is applied during the execution phase of the program. Dynamic code analysis involves executing parts of the code to check for any inconsistencies and defects while the program is running.

Automated generation of test suites enables the tester to analyse a larger proportion of the source program. This greatly reduces the time and effort required for manual inspection for test case generation.

Initially, an exceptional test plan is designed which is used to carry out the dynamic code analysis. The selected test cases and inputs to the dynamic code analyser determine the outcomes of the test. The test conditions are used to evaluate certain functionalities of the system. The main objective of this dynamic analysis is to eliminate all the potent defects from the software ahead of time, so as to elevate the performance rate of the product when disclosed in the market.

This section dealing with dynamic code analysis further bifurcates into two sections. The first part elucidates on the scope and tool selection that comprises of the overall sequence, inspection goals, and test conditions, followed by the selection of the dynamic code analysers. The second part throws light on the test plan that narrows down to the approach of test plan, the features to be tested and consequently the test cases.

Both static and dynamic analysis go hand-in-hand. The output of static analysis could provide suggestions for dynamic analysis while the output of dynamic analysis can be used as feedback information for static analysis [9].

# **INSPECTION GOALS**

Studies revealed that the standard practices were adding a lot of complexity and bureaucracy to the team members in the case of medium and small sized projects that spanned up to 9 months duration. Here are some of the inspection goals that we developed for dynamically analysing the code for this open source game of Sudokuki. We made sure that most of the goals are non-trivial as it would deepen the knowledge about the testing process and even in Java itself.

The goals are as follows:

Performance Defects: It includes various code anomalies that affect the performance of the software such as inefficient code leading to increased computational costs, unnecessary operations within a loop statement. Moreover, we also need to explore using alternative data structures and algorithms for enhancing performance [9].

Incorrect Operations: Operations that are rendered invalid during execution fall under this category. E.g. “division by zero, illegal function calls, freeing de-allocated memory” [9].

Dead Code and Data: It is a common runtime defect, where the code fails due to unreachable or unused data. Mistakes in logic can lead to dead code [9].

Incomplete Code: Inconsistent branching statements, use of variables without proper initialization. It may also include absence of proper termination and race conditions leading to deadlock situation [9].

1. **SELECTING THE DYNAMIC ANALYSING TOOL**

Dynamic code analysis as discussed above is the inspection of the code of a software during its execution. There are several steps involved in this process. They are

1. Input preparation

2. Launching the test program

3. Acquiring required parameters

4. Interpreting the output.

Hence, it is very crucial for the testers to carefully select the code analysing tools keeping in mind the kind of inspection goals they have set up to achieve. Rather than selecting a number of tools to test the application we here select two tools that closely address the inspection goals that we the authors have set up. For us to select these two tools, we had to filter from a number of dynamic code analysing tools available online. The filtering has been done as follows. We initially found about 6 tools that are closely related to our area of focus. Purify is a dynamic testing tool developed by IBM. It is not an open source software, hence this tool is out of scope for us to use in this task of analysing the Sudokuki application. Next, while considering TestGen4J, its main focus is to perform a boundary value testing. This doesn’t match with the inspection goals we have set up, hence this tool has been rejected from the list of tools. With the case of JUnit Test Case Builder (JUB) even though it generates test cases for the java class files it requires all the test cases to be stored in the source code repository [13].

Eventually, we ended up with two tools namely the CodePro Analytix and Randoop. EvoSuite is another test case generating tool, but the reason why we were not interested with it was that it requires all the generated test cases to be stored in the source code repository that is administered by the IDE that the programmer is using to perform the analysis. Getting into the details CodePro Analytix is a Dynamic code analysis tool that has been developed by Google [14]. This tool generates test case for all the java classes the application being tested contains. It provides the users with features like Code Metrics, Code Coverage, Code Analysis and Unit test generation [14]. Randoop is another tool similar to that of CodePro which generates JUnit format test cases. “Feedback-directed random test generation” is the concept is that is involved in this tool [15]. These tools have been keenly looked into if the test cases they are generating are in relation with the inspection goals. Study has been done regarding this and the general information about the tools have been provided in the Appendix below.

1. **EXECUTION OF DYNAMIC ANALYSIS**

In this section discusses how the dynamic analysis of the source code is done. Here we will follow a series of steps in order to reach our goal. Considering the time and knowledge factors ample study has been done in this area and the process has been carried out transparently. The whole process is carried out in three different phases. They are:

1. Basic approach of how the test is carried out
2. What features have to be tested
3. Generating the Test Cases

In phase 1 we initially discussed about what kind of goals should be set up. As in dynamic analysis we are concerned with that errors that pop up when a particular application is being executed. Hence, the testers need to address such issues with thorough inspection and make sure such errors don’t prevail in the system.

Once the goals have been set up, now we decided with the test plan i.e. the series of steps that are to be executed to generate the test cases. Next, we have executed the dynamic tools that were opted against the source code and generated the test cases. Finally, these test cases would be analysed against the inspection goals and verified and eventually resolved.

In phase 2, we will looked into to what kind of issues do we actually need to look into? Here in the case of Sudokuki, one possible situation is when a user is entering a number in the grid and this happens to be a wrong number and this goes on until the whole grid is finished and finally the win message pops up. Such issues need to be resolved as this would affect the basic functionality of the game. During the testing process there are several risks the testers might run into, hence certain risk mitigation strategies must also be set up as a contingency plan. By far as we have observed executing the tools CodePro Analytix and Randoop. Best results showed up when each and every package of the application are tested individually. Finally in this phase the code is run against the tools selected and the corresponding readings are noted and further analysis has to be made.

Finally, in this phase both the tools one at a time will be executed against the source code of the application and have the test cases generated. These test cases are then run accordingly as unit test for CodePro Analytix and regression testing for Randoop. Then the readings of what the errors and failures of each test run are noted and further analysis is done.

1. **REFLECTION ON OUTCOME**

Here in this report we will mainly concentrate on comparing the tools that we have selected to perform the dynamic code analysis. The tools are CodePro Analytix and Randoop.

After conducting dynamic analysis on the given source program, we arrived at the following findings. The possible factors leading to these observations have been explored below:

1. The number of test cases generated in case of Randoop were significantly higher when compared to CodePro Analytix.
2. There was as striking contrast between the number of errors generated by Randoop and CodePro Analytix. Randoop generated very few errors on the whole while errors generated by CodePro Analytix were noticeable.
3. The failures were generated at a consistent rate by CodePro Analytix while they were relatively inconsistent in Randoop.
4. The time taken for test case generation and also detecting failures by Randoop was observed to be much higher than Codepro Analytix.

**Factors affecting the outcome of both tools:**

Randoop uses “Feedback-directed random test generation” approach. It is essentially an iterative process of test generation where the previously generated tests form the basis for the next iteration. The computations from the previous text generation are used to update the algorithm for test case generation. Further, the sample space for tests is pruned [17].

Thus, the chance of generating illegal and redundant test cases is substantially lowered going further into the process [17]. This explains why Randoop takes more time. The inconsistency of failure generation can also be attributed to this. Further, it uses regression testing approach which aims at generating varied test cases. This leads to more error detection.

CodePro follows a “skeleton approach” for test case generation. This results in low “mutation score” and “code coverage.” This is the reason behind the significantly lower number of test cases generated by CodePro. CodePro follows a deterministic approach for test case generation and has better test case generation ability. It generates 16.4% more test cases than manual testing [18].

.

1. **CONCLUSIONS**

Dynamic analysis is the process of observing the behavior of a software program through its execution on sample inputs [9]. The tools that are used to carry out the dynamic code analysis are instrumental on the client-side containing analysis code [6]. In dynamic analysis, testing process finds the bugs later on in the development process as it involves bugs on the executable code.

Software tools use automation to simplify the inspection and code review processes. The tools that have been chosen based on their exclusive performance to implement dynamic analysis are Randoop and CodePro. A comparative study between them shows us that for a certain number of generated test cases for a particular package, the corresponding error rate and number of failures is zero in most cases in Randoop. On the other hand, the generated test cases in CodePro reflect a higher rate of errors and failures. This inspective study, indicates higher efficiency of Randoop over CodePro Analytix in performing dynamic analysis of the software. The feedback for static analysis can be foreseen using the outputs of dynamic analysis [9]. Thus, dynamic analysis confers a significant aid in binding the structural and functional parts of the software [16].

**REFERENCES**

[1] Hung-Hsuan Lin; I-Chen Wu, "Solving the Minimum Sudoku Poblem," *Technologies and Applications of Artificial Intelligence (TAAI), 2010 International Conference on* , vol., no., pp.456,461, 18-20 Nov. 2010

[2] book

[3] “FindBugsTM - Find Bugs in Java Programs.” [Online]. Available: http://findbugs.sourceforge.net/. [Accessed: 14-Apr-2015].

[4] Rutar, N.; Almazan, C.B.; Foster, J.S., "A comparison of bug finding tools for Java," *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on* , vol., no., pp.245,256, 2-5 Nov. 2004

[5] “PMD.” [Online]. Available: http://pmd.sourceforge.net/. [Accessed: 14-Apr-2015]

[6] C. Chahar, V. S. Chauhan, and M. L. Das, “Code analysis for software and system security using open source tools,” *Inf. Secur. J. Glob. Perspect.*, vol. 21, no. 6, pp. 346–52, 2012.

[7] IEEE Std. 1028-1997, *IEEE Standard for Software Reviews*, clause 38.

[8] “FindBugs Bug Descriptions.” [Online]. Available: http://findbugs.sourceforge.net/bugDescriptions.html. [Accessed: 15-Apr-2015].

[9] A. K. Ashish and J. Aghav, “Automated techniques and tools for program analysis: Survey,” in *2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*, 2013, pp. 1–7.

[10] “FindBugs Bug Descriptions.” [Online]. Available: http://findbugs.sourceforge.net/bugDescriptions.html. [Accessed: 15-Apr-2015].

[11] “Find Java concurrency bugs with ThreadSafe.” [Online]. Available: http://www.contemplateltd.com/threadsafe. [Accessed: 14-Apr-2015].

[12] Loveland, S., "Using Open Source Tools to Prevent Write-Only Code," *Information Technology: New Generations, 2009. ITNG '09. Sixth International Conference on* , vol., no., pp.671,677, 27-29 April 2009.

[13] Shuang Wang; Offutt, J., "Comparison of Unit-Level Automated Test Generation Tools," *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on* , vol., no., pp.210,219, 1-4 April 2009

[14] “CodePro Analytix User Guide - Java Developer Tools — Google Developers.” [Online]. Available: https://developers.google.com/java-dev-tools/codepro/doc/. [Accessed: 15-Apr-2015].

[15] “randoop - Random test generation - Google Project Hosting.” [Online]. Available: https://code.google.com/p/randoop/. [Accessed: 15-Apr-2015].

[16] Benedusi, P.; Benvenuto, V.; Tomacelli, L., "The role of testing and dynamic analysis in program comprehension supports," *Program Comprehension, 1993. Proceedings., IEEE Second Workshop on* , vol., no., pp.149,158, 8-9 Jul 1993

[17] Robinson, B.; Ernst, M.D.; Perkins, J.H.; Augustine, V.; Nuo Li, "Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs," *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on* , vol., no., pp.23,32, 6-10 Nov. 2011

[18] Kracht, J.S.; Petrovic, J.Z.; Walcott-Justice, K.R., "Empirically Evaluating the Quality of Automatically Generated and Manually Written Test Suites," *Quality Software (QSIC), 2014 14th International Conference on* , vol., no., pp.256,265, 2-3 Oct. 2014

Table 4: Selecting the Dynamic code analyzer:

|  |  |  |  |
| --- | --- | --- | --- |
| S.NO | Dynamic Code Analysis Tools | Decision | Motivation |
| 1. | Purify | Reject | * It is a memory debugger program used by software developers * It is mainly used to detect memory access errors in programs especially those written in c or c++ rather than for Java in this case [purify site] |
| 2. | TestGen4J | Reject | * It automatically generates JUnit test cases from java class files. * Its main focus is to perform boundary value testing of the arguments passed to methods. * It uses rules written in a user-configurable XML file [Comparison of Unit-Level Automated Test Generation Tools] |
| 3. | JUB(JUnit test case Builder) | Reject | * It is a test case generator framework accompanied by a number of IDE specific extensions. * It has hardwired dependencies on properties files in Builder [Comparison of Unit-Level Automated Test Generation Tools] |
| 4. | Google CodePro Analytix | Accept | * Dynamic code analysis tool that offers features like code metrics, code analysis, unit test case generation and code coverage [site] * Errors could be rectified on the go using this tool. * Facilitates the user to automate the creation of the test cases |
| 5. | Randoop | Accept | * Automatically generates test cases for java classes using feedback-generated random test generation [Randoop: Feedback-Directed Random Testing for Java] * “Achieved higher block and predicate coverage than model checking” [Randoop: Feedback-Directed Random Testing for Java] |
| 6. | EvoSuite | Reject | * It automatically produces Junit test suites for a given java class * It requires java byte of the class under test, along with its dependencies |

Table 5: Properties of CodePro Analytix:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | RANDOOP TESTING | | | CODEPRO ANALYTIX TESTING | | |
| **Package Name** | **Tests** | **Errors** | **Failures** | **Tests** | **Errors** | **Failures** |
| Gnu | - | - | - | 10/10 | 10 | 0 |
| gnu.gettext | - | - | - | 20/20 | 20 | 0 |
| net.jankenpoi | - | - | - | 53/53 | 4 | 1 |
| net.jankenpoi.i18n | 5 | 0 | 0 | 47/47 | 1 | 0 |
| net.jankenpoi.sudokuki | 7353 | 0 | 0 | 33/33 | 3 | 1 |
| net.jankenpoi.sudokuki.controller | - | - | - | 48/48 | 4 | 2 |
| net.jankenpoi.sudokuki.generator | 2 | 0 | 0 | 17/17 | 0 | 0 |
| net.jankenpoi.sudokuki.generator.suexg | - | - | - | 1/1 | 0 | 0 |
| net.jankenpoi.sudokuki.model | 314 | 0 | 23 | 100/100 | 6 | 9 |
| net.jankenpoi.sudokuki.preferences | 1493 | 0 | 160 | 10/10 | 0 | 0 |
| net.jankenpoi.sudokuki.resources | 1 | 0 | 0 | - | - | - |
| net.jankenpoi.sudokuki.solver | 20788 | 0 | 0 | 59/59 | 37 | 0 |
| net.jankenpoi.sudokuki.ui | 34 | 0 | 0 | 1/1 | 0 | 0 |
| net.jankenpoi.sudokuki.ui.swing | - | - | - | 39/103 | 11 | 2 |
| net.jankenpoi.sudokuki.ui.text | - | - | - | 22/22 | 0 | 0 |
| net.jankenpoi.sudokuki.view | - | - | - | 31/31 | 1 | 7 |
| net.sourceforge.plantuml | - | - | - | 1/27 | 0 | 0 |
| net.sourceforge.plantuml.sudoku | 21 | 20 | 0 | 3/54 | 0 | 1 |