International Journal of Applied Engineering Research ISSN 0973-4562 Volume 12, Number 24 (2017) pp. 15202-15214
© Research India Publications. http://www.ripublication.com

A Severity Based Source Code Defect Finding Framework and Improvements over Methods

K Venkata Ramana
Assistant Professor, Department of Computer Science & Engineering,
Vallurupalli Nageswara Rao Vignana Jyothi Institute of Engineering (VNR), Hyderabad, Telangana, India.

Orcid Id: 0000-0002-2208-7966

Dr K Venugopala Rao
Professor, Department of Computer Science & Engineering,
G.Narayanamma Institute of Technology and Science, Hyderabad, Telangana, India.

Abstract

Application development industry always looked for defect free application as most of the errors can be detected during the compilation time, but defects are only to be noticed during the execution time or during the production. Once detected the application with defects needs to go through most of the development life cycle phases for correction. This brings a higher cost and delay in the code development process. Hence, the demand from the application development industry to the researchers is to find processes or methods or algorithms or frameworks to identify the defects in the application source code during the development process. The goal is to build an automated framework for detection of the defects as the manual process is prone to errors due to the insufficient knowledge of the developers. There are various research attempts made to automate the process as a framework. Nonetheless, most of the outcomes from those researchers are resulted into overlapped defect detection and some of the reported defects are considered to be false positive. Thus, making those processes far from correct. In this paper, the comparative study of the popular automated frameworks is presented and compared with the proposed novel framework. The proposed method detects the defects with the given priority to solve called severity. The results demonstrate significant improvements in terms of accuracy over the other methods.

Keywords: PMD, JLINT, Bandera, FindBugs, Defect Severity, Complexity reduction.

INTRODUCTION

Motivated by the demand from application programming and the demand for high performing applications from the consumers, the researchers are motivated towards finding the automated framework to detect defects during the application development phase rather in the production or intense testing. Many research attempts are carried out in order to detect the defects by various individual researchers or research organizations. The most popular methods for defect detections are pattern machine techniques, data flow analysis, type machining, model validation or theorem verifications.

Majority of the tools or frameworks demonstrate similar type of results where the type of the bugs are not well defined and demands a non over lapping measure of the defect metric. Henceforth the demand for the newer methods or processes or the framework cannot be ignored. In the process of the study in this research, various short comings are identified and can be used for enhancement. The work by Nick Rutaret al. [1] demonstrates a similar comparative analysis and defined the following areas to be addressed:

- A detailed comparison of the defect detection processes or methods or frameworks are to be carried out
- Various tools measure defects depending on various models. Hence, the results are varying and overlapping resulting into non-standard detection metrics.
- None of the single tools detects all possible defects in the source code. Thus there is a need to gather all possible tools to merge the results and provide a single result for each analysis.

Understanding the need and the well-defined areas, this research addresses all the areas. The outcomes are the effect of the collaborated research and furnished here:

- The comparative analysis based on a proposed metric model is formulated by K Venkata Ramana et al [2].
- A novel metric for the detection of the defects are proposed and tested on standard java source codes in the work by K Venkata Ramana et al. [3].
- Finally and conversely, as per the proposal by Nick Rutar et al. [1], this work considers creating a novel framework to detect the defects using a severity based measure.

The severity metric is also been proposed during the course of study by K Venkata Ramana et al. [4].

Henceforth, this work performs a deep analysis of the performance and establishes the thoughts from the claim to improve the performance.

The rest of the work is furnished such that in section–II, the overview of the outcomes are been analysed, in section–III, the already proposed framework and metric is reconsidered for comparison, in section – IV the proposed algorithm is furnished, in section – V the defect detection metrics are considered in order to establish the relevance of the similarity, in the section – VI the comparative analysis between the frameworks are discussed, in section – VII, the results are been discussed and finally the work presents derived conclusion in the section – VIII.

Outcomes from the parallel researches

In this section of the work, the most popular frameworks and tools for detection of bugs are discussed from the developer's perspective. The major different between the frameworks is few of the frameworks consider the actual source code for analysing the defects and few of the frameworks consider binary code or the compiled jar files for the detection of defects.

This work considers few frameworks from both the categories in order to justify the claim of improvement in accuracy and time complexity.

A. PMD

PMD or popularly known as Project Monitoring Directives framework is a syntactic checker on the application source code. The PMD does not reply on the data flow graphs or data flow component [5]. PMD provides static detection of the defects in the source code, where certain defects are based on the situation of the development criteria. Nevertheless, researches define that PMD detects some of the defects which are false positive.

The architecture of the PMD [Fig – 1] is elaborated by Arthur Carroll [6]

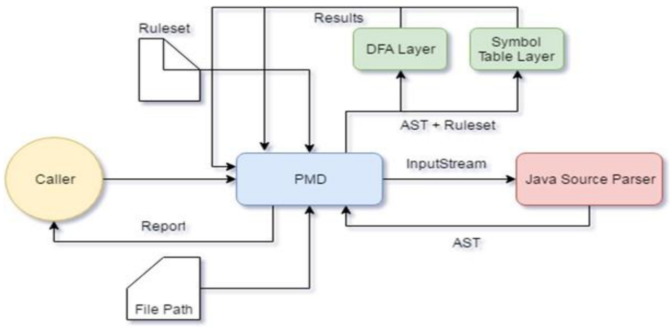

Figure 1: PMD Architecture – High Level Building Blocks

B. Bandera

Bandera [7] is a defect detection framework and analyses the source code by model verification and abstraction analysis. The Bandera framework can be engaged in the development phase like the other popular frameworks. In order to use the Bandera, the developers expected to define the custom model in terms of directives. In the absence of the customized directives, the framework analyses the deadlocks in the code along with standard verification models like path verification.

The architecture of the Bandera presented by J. C. Corbett et al [7] and furnished here [Fig – 2].

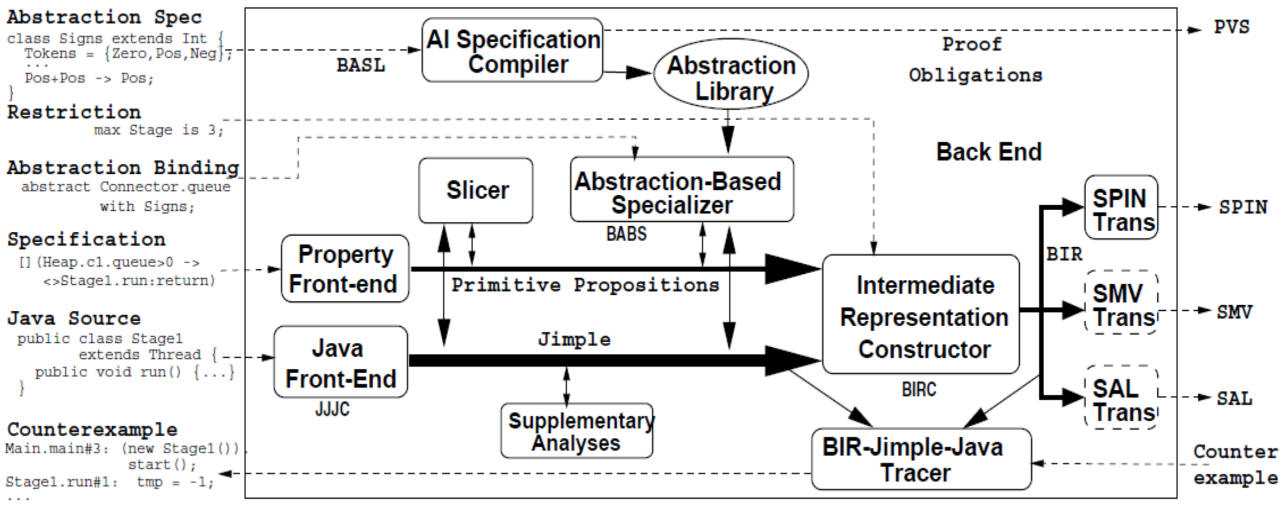
```
Abstraction Spec
class Signs extends Int {
  Tokens = {Zero,Pos,Neg};
  ...
  Pos+Pos -> Pos;
}
Restriction
          max Stage is 3;
Abstraction Binding
 abstract Connector.queue
         with Signs;
Specification
 [](Heap.c1.queue>0 ->
   <>Stage1.run:return)
Java Source
 public class Stage1
        extends Thread {
   public void run() {...}
 }
Counterexample
Main.main#3: (new Stage1())
                start();
Stage1.run#1:  tmp = -1;
...
```

Figure 2: Bandera Architecture

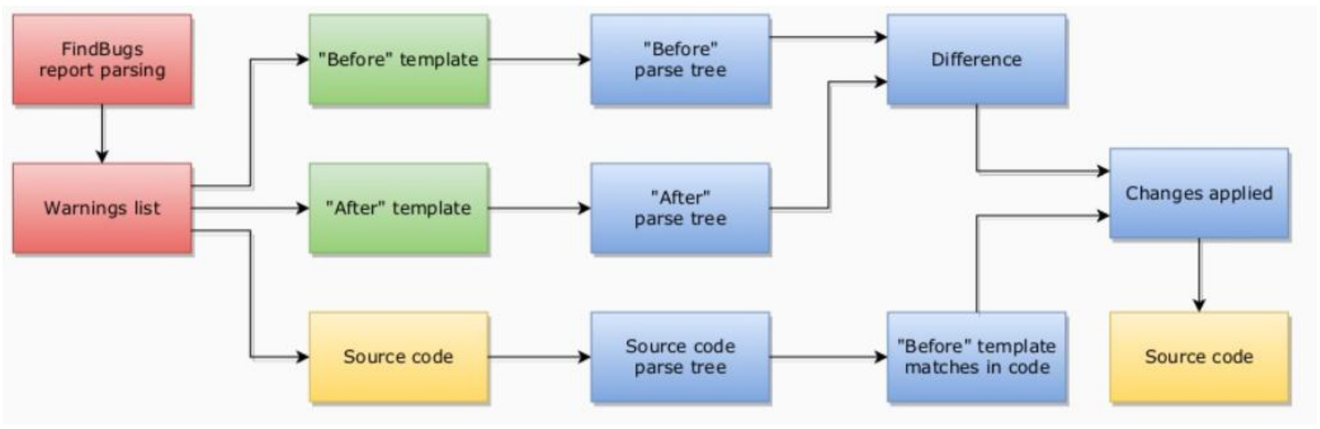

Figure 3: Architecture of FindBugs

C. FindBugs

Among the various defect detection frameworks, the FindBugs proposed by D. Hovemeyer et al. [8] is the most popular one. FindBugs uses the ad-hoc techniques to detect the efficiency of the code, reusability of the code and also the precision of the code. FindBugs also allows the developer to enhance capabilities by introducing customizable detector rules in the source code.

The architecture of the FindBugs, proposed by D. Hovemeyer et al. [8] is furnished here [Fig – 3].

Henceforth this work presents the novel framework and defect detection metric in the next section in order to analyse the performance difference.

Proposed Framework and Metric

In the previous work by K Venkata Ramana et al. [3] have proposed the novel framework. Majority of the research directions proposed by other researchers mentioned to combine the outcomes from popular research outcomes and present by removing the overlaps. This framework conversely proposes a novel metric for detection of the defects and based on the severity based defect report rejection method, detects the defects.

The framework is presented here [Fig – 4].

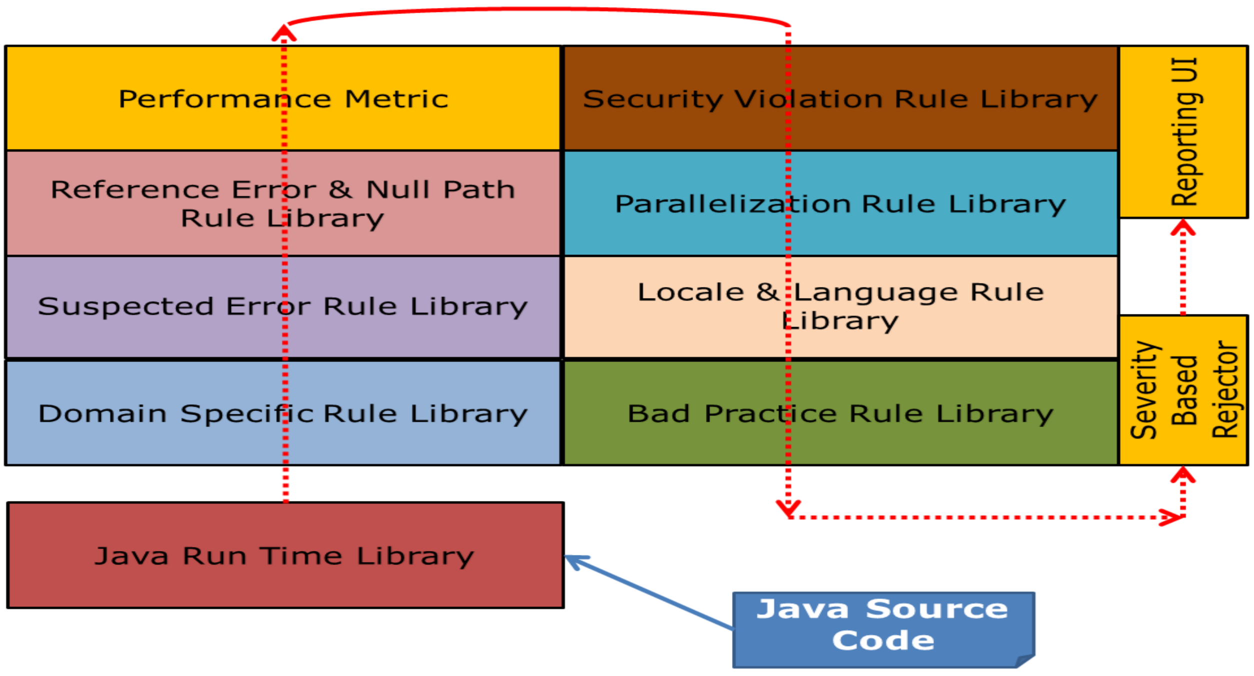
Performance Metric
Security Violation Rule Library
Reporting UI
Reference Error & Null Path Rule Library
Parallelization Rule Library
Suspected Error Rule Library
Locale & Language Rule Library
Severity Based Rejector
Domain Specific Rule Library
Bad Practice Rule Library
Java Run Time Library
Java Source Code

Figure 4: Architecture of Proposed Framework

Further, the novel defect detection metric is furnished here with the brief description [Table – 1].

Table I: Novel Defect Detection Metric

Serial Number	Parameter Name	Parameter Description
1	Domain Specific Coding Standards	Violation of the customer or organization or government policies
2	Bad Practices	Java coding standard mismatches
3	Suspected Errors	Potential defects due to logical errors in the source code
4	Locale Errors	Language or regional setup like date, currency or time format
5	Reference Errors	Overlapping of the class visibility
6	Parallelism Error	Tending to dead lock situation
7	Performance Errors	Deteriorate in popular performance factors
8	Security Errors	Potential exposure of the data structures to other non-member functions in the code
9	Null Path Errors	Potential code which tend to become unreachable

The logical relations of the defined metric parameters with the existing parameters are defined in other sections of the work.

The proposed framework deploys a severity based algorithm in order to report the defects. The algorithm is explained in the next section.

Proposed Severity Based Detection Algorithm

The existing algorithms are criticized due to the overlapping nature of the reported defect categories and due to the nature of the false positive reports.

Henceforth, in this work, with the light of the novel defined reporting matric the new algorithm is proposed.

The steps of the algorithm are furnished here:

Step -1. Read testProject.jar

Step -2. For each class in testProject.jar

Step -3. Open class n & Perform analysis Pass [1 to 9]

- ➢ Domain Specific coding Standard pass
- ➢ Read each entry in the declarative rule set

```
Declarative Rule Set:
EmployeeType="Salaried"; //Default
EmployeeAge=30; // Default – Minimum
EmployeeSalary=25000;//Default – Minimum
…………………………………..
```

- ▪ If each Variable name & Value do match with declarative rule
  - • Continue;
- ▪ Else, in case any mismatch
  - • Increase CS Count as CS = CS +1.
- ➢ End of DSCS Pass.

- ➢ Programming Bad Practices Pass
- ➢ Read each entry in the Practice rule set

```
Rule -1: Any Function / Method returning NULL value and return accepted in any variables for which default value is not NULL.
Change: Mark Return Statement

Rule – 2: Any default value initialization should be without constructor
Change: String data = new String("Hello"); - Should be marked
Change: String data = "Hello"; - Should not be marked

Rule – 3: Any blank initialization should be avoided
Change: ArrayList Employee = new ArrayList() – Should be marked

Rule – 4: The counter part of the exception handling should not be iterated.
………..
```

- ▪ If any rule is Violated
  - • Increase BPAS Count as BPAS = BPAS + 1
- ▪ Else, in case the rule is satisfied
  - • Continue;

- End of BPAS Pass.

- Suspected Errors Pass
- For each Decision Path in the collection, generate input

Input Variable	Desired Value	Value Generation Rule	Expectations
a	a > 50	a = rand() % 50	100% Decision Path Visited

a	a <= 50	a = 50
		a = rand() % 100
			Decision Path Visited
			Decision Path Visited for all values > 50

b	b < 10	b = rand() % 10	100% Decision Path Visited
….	….	….	….

  - If the decision path is traversed
    - Continue
  - Else, in case the decision path is Ignored
    - Increase the SE count as SE = SE +1
- End of SE Pass
- Locale Errors Pass
- For each Language option in the GUI
  - If date ( ) format mismatch
    - Increase the LE count as LE = LE +1
  - Else, If time ( ) format mismatch
    - Increase the LE count as LE = LE +1
  - Else, If Language library not available
    - Increase the LE count as LE = LE +1
  - Else
    - Continue ;
- End LE Pass.
- Reference Error Pass
- For each member in the class, check Reference Errors

Parent Access Rule	Valid Child Access Rule
public	Blank Default Public Protected Private
protected	Blank Default Protected Private
….	….

  - If Reference Error Override
    - Increase the RE Count as RE = RE +1
  - Else Continue.
- End of RE Pass.

- Parallelism Error Pass
- For each thread in the class, read the thread Rule Engine

Present State	Next State	Methods Expected in Order

Ready to Run	Running	init()
		run()

Ready to Run	Dead	stop()
		exit()

Running	Sleep	sleep()
Running	Wait	wait()

Running	Dead	stop()
		exit()

Wait	Read to Run	notify()
		notifyAll()

…	…	…

  - If Rule Engine violation
    - Increase the PE as PE= PE +1
  - Else, If Rule Order match
    - Continue
- End of PE

```
➢ Performance Error Pass
    ▪ If MaxHeap Ratio is equal to MinHeap Ratio
        • PerE = Per E +1
    ▪ If Native Heap is less than Thread Heap
        • PerE = Per E +1
    ▪ If code catch is greater than JavaHeap
        • PerE = Per E +1
    ▪ Else
        • Continue
➢ End of PerE Pass

➢ Null Path Pass
➢ For each exit section of every method
    ▪ If return NULL is encountered
        • If exit section Contains more than 1 line
            ♦ NPLE = NPLE +1
        • Else
            ♦ Continue
    ▪ Else
        • Continue
➢ End NPLE Pass.
```

Step -4. Close detector Pass

Step -5. Report all Variables based on the severity

Step -6. End of Detector.

Hereafter, this work analyses the algorithms in the further section. However, the proposed metric is to be justified with the existing metrics. In the next section of this work, the comparative study is proposed.

Comparative Metric Analysis

In this section of the work, the existing metric for defect detection [Table – 2] [1] is elaborated and the relevancy with the proposed metric is furnished [2] [Table – 3].

Table II: Existing Metric for Bug Detection

Serial Number	Parameter Name
1	Null Deference
2	Possible Dead Lock
3	Exception
4	Array
5	Divided by Zero
6	Unreachable Code
7	String Errors
8	Object errors
9	I/O Errors
10	Duplicate Statements
11	Design Error
12	Unnecessary return statements

Table III: Relevancy Mapping of Exisint to Proposed Metric

Proposed Metric Parameter Serial Number [Table – 1]	Existing Metric Parameter Serial Number [Table – 2]
9	1
6	2
2	3, 4, 5, 7, 10, 12
3	6
5	8,9
1	11
4	Additional Parameter
7	Additional Parameter
8	Additional Parameter

It is to be observed that, during comparison few of the proposed parameters are grouped, in order to reduce the overlapping of the defect information.

Further, the defect detection metric parameters are evaluated on three existing and one proposed framework [Table – 4].

Table IV: Existing Metric for Bug Detection

Serial Number	**Parameter Name**	**Available in**			
		FindBugs	**PMD**	**Bandera or JLINT**	**Proposed Framework**
1	Null Deference	**Yes**	**Yes**	**Yes**	**Yes**
2	Possible Dead Lock	**Yes**	**Yes**	**Yes**	**Yes**
3	Exception	No	No	No	**Yes**
4	Array	No	No	**Yes**	**Yes**
5	Divided by Zero	No	No	**Yes**	**Yes**
6	Unreachable Code	**Yes**	**Yes**	No	**Yes**
7	String Errors	**Yes**	**Yes**	**Yes**	**Yes**
8	Object errors	**Yes**	**Yes**	**Yes**	**Yes**
9	I/O Errors	**Yes**	No	No	**Yes**
10	Duplicate Statements	**Yes**	**Yes**	No	**Yes**
11	Design Error	**Yes**	No	No	**Yes**
12	Unnecessary return statements	No	**Yes**	No	**Yes**
Availability Count		8	7	6	**12**

Thus, the first claim of the research is justified as all the existing parameters are accommodated in the proposed framework and additionally few of the parameters are introduced to increase the accuracy.

In the next section of the work the existing algorithms are been compared with the proposed algorithm.

Comparative Framework and Algorithm Analysis

In this section of the work, the frameworks are been compared. The existing frameworks such as PMD, Bandera and FindBugs are compared with the proposed framework based on the usability [Table – 5] [1].

Table V: Exisiting and Proposed Framework Comparison

Comparison Factor	PMD	Bandera Or JLINT	FindBugs	Proposed Framework
Code Analysis Process	Static	Static	Static	Static
Analysis Input	Source Code	Source Code	Byte Codes / Compiled Codes	Byte Codes & Source Code
Available Interfaces	IDE Plugin, Command Line, GUI	Command Line, GUI	Command Line, GUI	Command Line, GUI
Methodology	Syntax Tree	Model Validation	Syntax Tree Analysis, Data flow graph analysis	Syntax Tree, Data Flow, Model Validation, Decision Path and Rule Based Analysis

Hence it is natural to understand that, the proposed framework enhancements the capabilities by using various models for defect detection. Nonetheless, the proposed framework does not provide the capabilities to integrate into any IDE unlike PMD.

Further, the existing algorithms are been analysed and compared with the proposed algorithm in order to justify the claim of improvements in the accuracy.

A. *PMD*

Firstly, the algorithm for the PMD framework is to be analysed. Thereafter, the drawbacks of the PMD algorithm are to be identified for further comparison.

PMD Algorithm [9]:

Step 1. Parse command line parameters

Step 2. Load rulesets/rules

Step 3. Determine languages

Step 4. Determine files

Step 5. Prepare the renderer

Step 6. Sort the files by name

Step 7. Check whether we can use the incremental analysis cache

a. Prepare the SourceCodeProcessor based on the configuration

b. Analyze the files.

c. Create input stream

d. Call source code processor

e. Determine the language

f. Check whether the file is already analyzed and a result is available from the analysis cache

i. Parse the source code. Result is the root AST node.

ii. Always run the SymbolFacade visitor. It builds scopes, finds declarations and usages.

iii. Run DFA visitor for building control flow graphs and data flow nodes.

iv. Run TypeResolution visitor

g. First run the rules that opted in for the rule chain mechanism

h. Run all the other rules and let them traverse the AST. The rules can use the symbol table, type resolution information and DFA nodes.

i. The rules will report found problems as RuleViolations.

j. Render the found violations into the wanted format

k. Store the incremental analysis cache

Step 8. Depending on the number of violations found, exit with code 0 or 4.

Flow Analysis

Further the flow analysis is been carried out

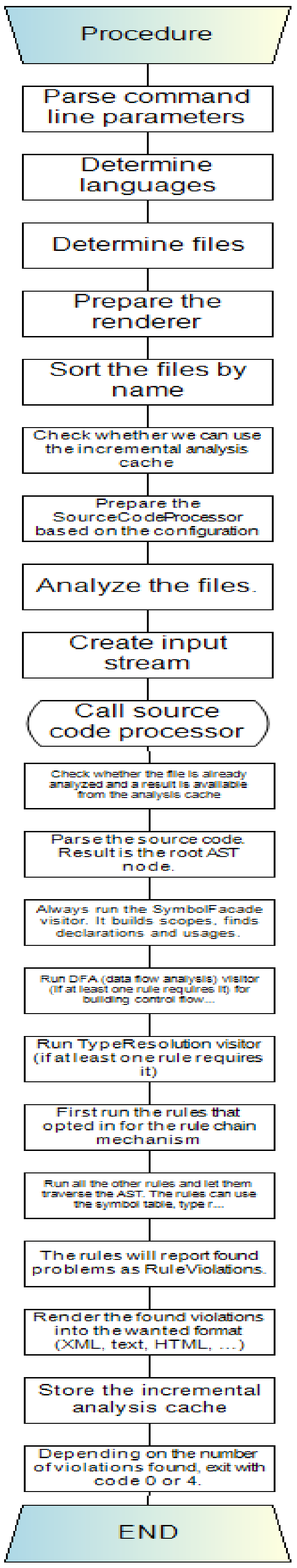
Procedure
Parse command line parameters
Determine languages
Determine files
Prepare the renderer
Sort the files by name
Check whether we can use the incremental analysis cache
Prepare the SourceCodeProcessor based on the configuration
Analyze the files.
Create input stream
Call source code processor
Check whether the file is already analyzed and a result is available from the analysis cache
Parse the source code. Result is the root AST node.
Always run the SymbolFacade visitor. It builds scopes, finds declarations and usages.
Run DFA (data flow analysis) visitor (if at least one rule requires it) for building control flow....
Run TypeResolution visitor (if at least one rule requires it)
First run the rules that opted in for the rule chain mechanism
Run all the other rules and let them traverse the AST. The rules can use the symbol table, type r....
The rules will report found problems as RuleViolations.
Render the found violations into the wanted format (XML, text, HTML, ...)
Store the incremental analysis cache
Depending on the number of violations found, exit with code 0 or 4.
END

Figure 5: Flow Analysis of PMD Algorithm

Drawbacks Identified:

A number of research attempts are been made to identity the shortcomings for further improvements. Few highly rated points are been furnished here [1]:

- Most of the detections are considered suspicious under certain situations.
- The number of defects detected depends on the number of rule sets deployed for the detection, which is not standardised.

B. *Bandera*

Secondly, the algorithm for the Bandera framework is to be analysed. Thereafter, the drawbacks of the Bandera algorithm are to be identified for further comparison.

Algorithm [7]:

```
Step 1.   Generate the finite state model
Step 2.   Generate Linear Temporal Logic
Step 3.   Generate Computation Tree Logic
Step 4.   Deploy Model Checker
              a. Start SPIN
              b. Start dSPIN
              c. Start SMV
Step 5.   Report the problem
Step 6.   Stop Analyser
```

Flow Analysis

Further the flow analysis is been carried out[6]

Drawbacks Identified:

A number of research attempts are been made to identity the shortcomings for further improvements. Few highly rated points are been furnished here [1]:

- Does not provide the detection on realistic java programs on version 0.3b2.
- Multiple model for the detection, hence increasing the change to increase the overlapping

C. *FindBugs*

Further, the algorithm for the FindBugs framework is to be analysed. Thereafter, the drawbacks of the FindBugs algorithm are to be identified for further comparison.

Algorithm [10]:

```
Step 1.   Detector: Find hash equals mismatch
              a. Defines a co-variant version of the equals() or compareTo() method.
              b. The Object.equals() version of the method will be used.
Step 2.   Detector: Return value of method ignored
              a. This detector looks for places in your code where the return value of a method is ignored when it shouldn't be.
Step 3.   Detector: Null pointer dereference and redundant comparisons to null
              a. This detector looks for two types of problems.
              b. Detects a similar problem when it's able to determine that one of the values is null
Step 4.   Detector: Field read before being initialized
              a. This detector finds fields that are read in constructors before they're initialized.
Step 5.   Report the problem
Step 6.   Stop Analyser
```

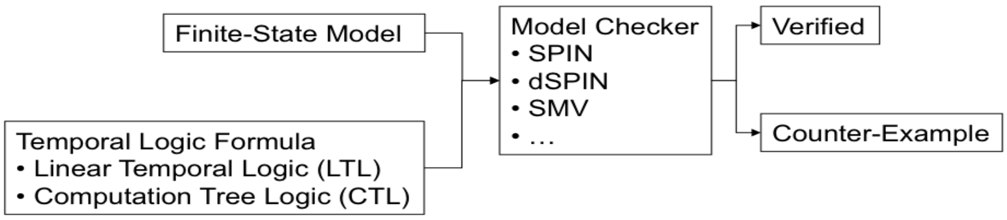

Figure 6: Flow Analysis of Bandera Algorithm

Flow Analysis

Further the flow analysis is been carried out [7]

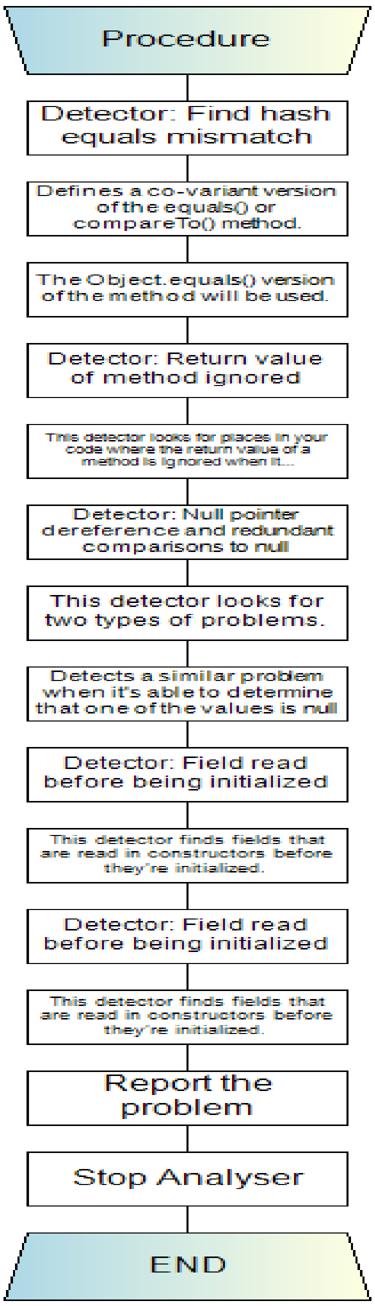
Procedure
Detector: Find hash equals mismatch
Defines a co-variant version of the equals() or compareTo() method.
The Object.equals() version of the method will be used.
Detector: Return value of method ignored
This detector looks for places in your code where the return value of a method is ignored when it...
Detector: Null pointer dereference and redundant comparisons to null
This detector looks for two types of problems.
Detects a similar problem when it's able to determine that one of the values is null
Detector: Field read before being initialized
This detector finds fields that are read in constructors before they're initialized.
Detector: Field read before being initialized
This detector finds fields that are read in constructors before they're initialized.
Report the problem
Stop Analyser
END

Figure7: Flow Analysis of FindBugs Algorithm

Drawbacks Identified:

A number of research attempts are been made to identity the shortcomings for further improvements. Few highly rated points are been furnished here [11]:

- Serious lag in detection sophisticatedtechniques
- Complex process to identify defects due to false positive and true negative overlapping

D. *Proposed Algorithm Flow Analysis*

Finally, the algorithm for the proposed method is analyses and benefits are been identified [Fig – 8].

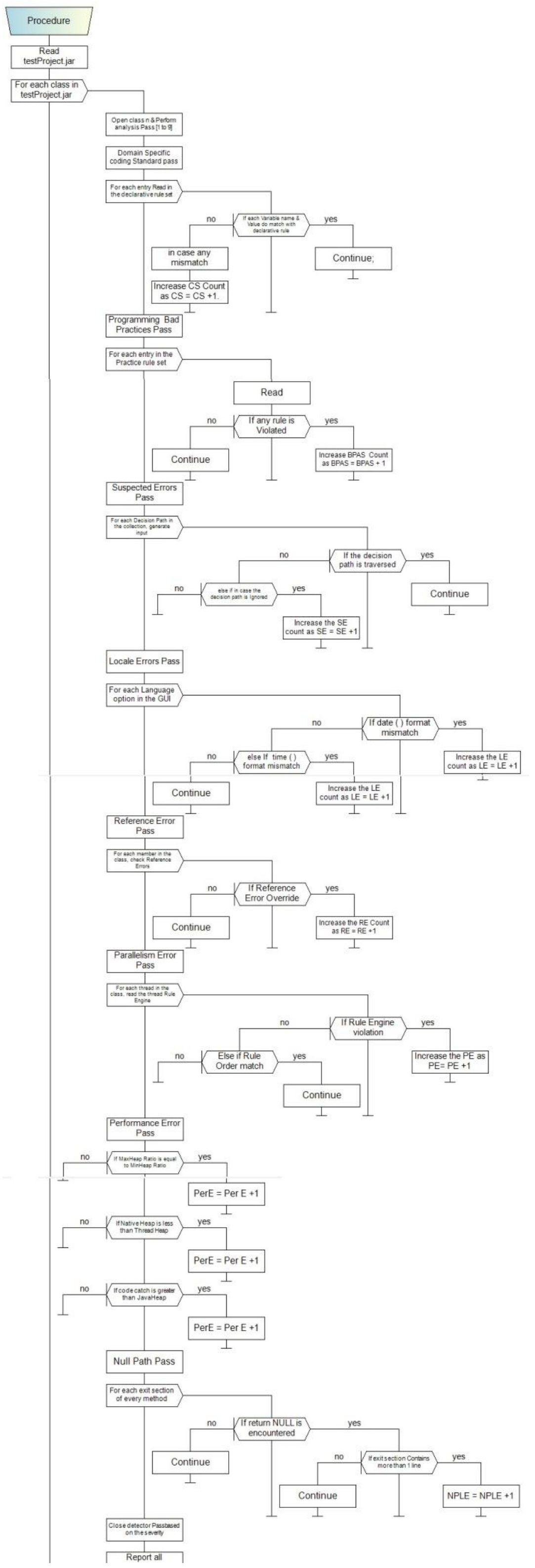
Procedure
Read testProject.jar
For each class in testProject.jar
Open class n & Perform analysis Pass [1 to 8]
Domain Specific coding Standard pass
For each entry Read in the declarative rule set
If each Variable name & Value do match with declarative rule
no
yes
in case any mismatch
Increase CS Count as CS = CS +1.
Continue;
Programming Bad Practices Pass
For each entry in the Practice rule set
Read
If any rule is Violated
Continue
Increase BPAS Count as BPAS = BPAS + 1
Suspected Errors Pass
For each Decision Path in the collection, generate input
If the decision path is traversed
else if in case the decision path is ignored
Increase the SE count as SE = SE +1
Continue
Locale Errors Pass
For each Language option in the GUI
If date ( ) format mismatch
else If time ( ) format mismatch
Increase the LE count as LE = LE +1
Continue
Increase the LE count as LE = LE +1
Reference Error Pass
For each member in the class, check Reference Errors
If Reference Error Override
Continue
Increase the RE Count as RE = RE +1
Parallelism Error Pass
For each thread in the class, read the thread Rule Engine
If Rule Engine violation
Else if Rule Order match
Continue
Increase the PE as PE= PE +1
Performance Error Pass
If MaxHeap Ratio is equal to MinHeap Ratio
PerE = Per E +1
If Native Heap is less than Thread Heap
PerE = Per E +1
If code catch is greater than JavaHeap
PerE = Per E +1
Null Path Pass
For each exit section of every method
If return NULL is encountered
Continue
If exit section Contains more than 1 line
Continue
NPLE = NPLE +1
Close detector Passbased on the severity
Report all

Figure 8: Flow Analysis of Proposed Algorithm

Identified Advantages:

The following features of the proposed algorithm are the benefits over three existing comparative models [Table – 6].

Table VI: Benefits of the Proposed Algorithm

Benefits of the Proposed Method	Problems in the Existing Method		
Proposed Framework	PMD	Bandera	FindBugs
Severity Based Measures	False positive detections	Multiple models for detection	False positive detections
Unique Non-Overlapping Metric	No standard rule sets		
Analysis on Java Binary Codes		No support for Java codes	
Symmetric detection of the bugs			Complex model for the detection

Hence, it is clear that the claims made by this research can overcome the problems of popular frameworks for defect detection.

Thus in the next section this work analyses the accuracy of the proposed algorithm and compares with the existing algorithms.

RESULTS AND DISCUSSION

The automatic frameworks are designed to provide the optimal metric for all types of bug detection and provide accuracy in the detection process.

Hence, in this section the work presents the metric reports and the accuracy for the defect detection.

A. *Metric Ranking Analysis*

During the course of studying the existing framework metrics and the proposed metric, based on the number of type of defect detection capabilities the frameworks are given weightages. The weight calculation process is simple as if the defect type can be detected, then that methodology gets a point and the process increments the weightage value. The findings are furnished here [Table –7].

Table VII: Ranking analysis

	FindBugs	**PMD**	**Bandera or JLINT**	**Proposed Framework**
Availability Count	8	7	6	**12**
Rankings	2	3	4	1

The results are analysed visually [Fig – 9].

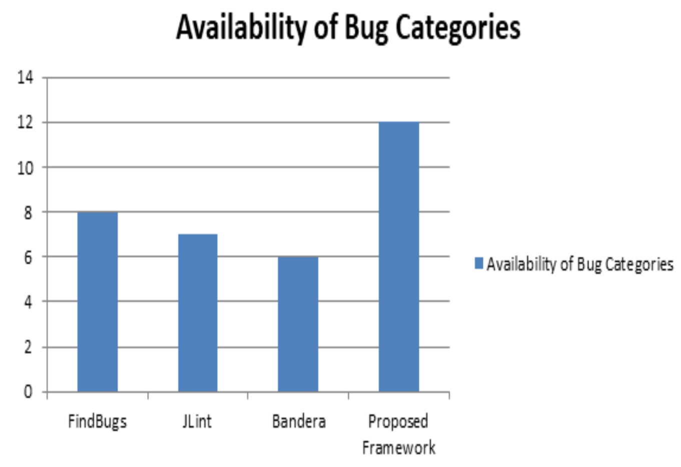

Figure 9: Ranking Analysis

Hence, it is natural to understand that the proposed framework provides the highest number of bug categories, thus ranked one in the analysis.

B. *Detection Accuracy*

Apart from the overall detection numbers, this work also furnishes the defect category wise number of defect detection [Table – 8].

The proposed method is capable of detecting the defects as per the proposed optimal framework.

The results are analysed visually [Fig – 10].

Table VIII: Detection Results in Proposed Framework

Name of Testing Data	Domain Specific Coding Standards	Bad Practices	Suspected Errors	Locale Errors	Reference Errors	Parallelism Error	Performance Errors	Security Errors	Null Path Errors	**Total**
JBoss 3.2.3	22	253	279	33	108	31	85	23	3	837
ArtOfIllusion	144	54	592	19	175	27	383	0	1	1395
Tomcat	9	61	102	44	86	16	77	0	10	405

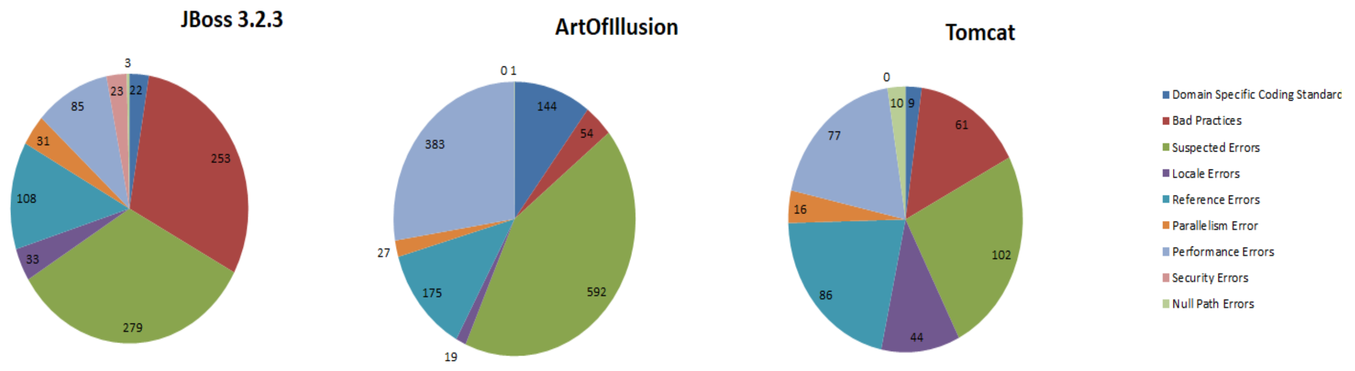

Figure 10: Detection Results in Proposed Framework

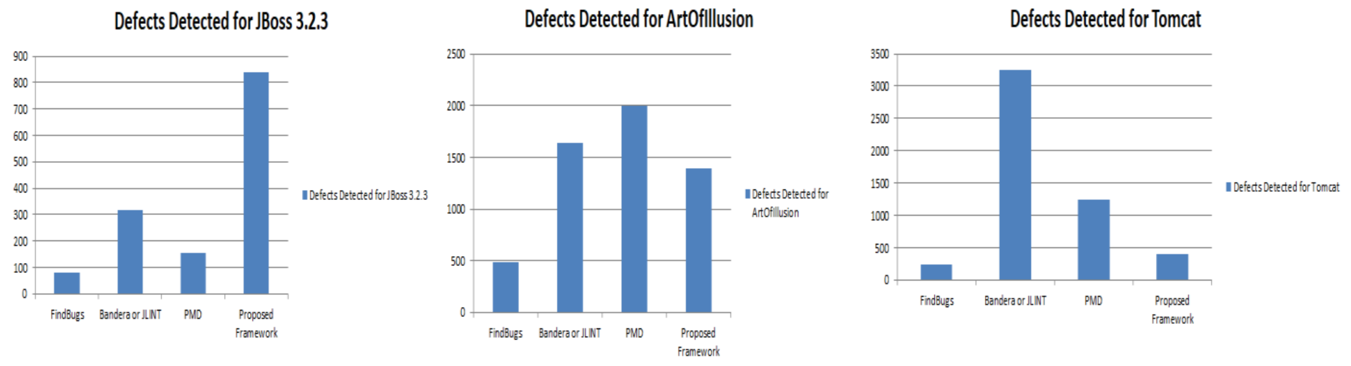

Figure 11: Defect Detection by all frameworks

C. *Detection Accuracy*

The accuracy of the frameworks is expected to be the highest and provide optimal number of defects without the false positive and the overlapped defects. Hence, in this section the work provides the accuracy of defect detections [Table – 9].

Table IX: Comparative detection Results

Name of Testing Data	**FindBugs**	**Bandera or JLINT**	**PMD**	**Proposed Framework**
JBoss 3.2.3	79	317	153	837
ArtOfIllusion	481	1637	1992	1395
Tomcat	245	3247	1236	405

Thus it is natural to understand that,

1) The proposed framework significantly reduces the overlapped defect reports and provides moderately high accuracy compared to findBugs.
2) The framework also reduces the overlapped and false positive defect reports to a high extend compared to JLintor Bandera and PMD.

The results are also visualized graphically [Fig – 11].

Thus with the knowledge of the study and the obtained results, this work presents the conclusion in the next section.

CONCLUSIONS

The objective of this research is to create an automatic defect detection framework to detect the defects in the application and reduction of the overlapping defect categories. In the course of demonstration, this work establishes a novel severity based defect metric by removing the overlapping framework and improves the detection rate. The highly satisfactory outcome from this work is obtained by deploying multiple models of defect detection mechanism such as Syntax Tree, Data Flow, Model Validation, Decision Path and Rule Based Analysis. This proposed framework is certainly an improvement in this domain of research and will create a higher impact in reducing the development time by providing defect free applications in the industry, academia and science.

REFERENCES

[1] Nick Rutar, Christian B. Almazan and Jeffrey S. Foster, "A Comparison of Bug Finding Tools for Java", ISSRE '04 Proceedings of the 15th International Symposium on Software Reliability Engineering Pages 245-256. November 02 - 05, 2004

[2] K Venkata Ramana and Dr K Venu Gopala Rao, "Investigation of Source Code Mining using Novel Code Mining Parameter Matrix:Recent State of Art", International Journal of Latest Trends in Engineering and Technology. Vol.(7)Issue(3), pp. 088-097

[3] Venkata Ramana Kaneti, "A Novel Automatic Source Code Defects Detection Framework and Evaluation on Popular Java Open Source APIs", International Journal of Advanced Research in Computer Science. ISSN No. 0976-5697

[4] K Venkata Ramana and Dr K Venugopala Rao, "An Evaluation of Popular Code Mining Frameworks through Severity Based Defect Rule",International Journal of Emerging Technology and Advanced Engineering. Volume 7, Issue 6, June 2017.

[5] PMD/Java. http://pmd.sourceforge.net.

[6] Arthur Carroll, "STATIC CODE ANALYSIS". URL - http://slideplayer.com/slide/10978396/

[7] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In Proceedings of the 22nd International Conference on Software Engineering, pages 439–448, Limerick Ireland, June 2000.

[8] D. Hovemeyer and W. Pugh. Finding Bugs Is Easy. http://www.cs.umd.edu/˜pugh/java/bugs/docs/findbugsPaper.pdf, 2003.

[9] How PMD Works, September 2017, ©2017 PMD Open Source Project. All rights reserved. http://pmd.sourceforge.net/snapshot/pmd_devdocs_how_pmd_works.html

[10] Chris Grindstaff, "Improve the quality of your code", IBM Developer Works, May 25, 2004. https://www.ibm.com/developerworks/library/j-findbug1/

[11] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh, "Experiences Using Static Analysis to Find Bugs",IEEE Software (Volume: 25, Issue: 5, Sept.-Oct. 2008)