Java Code Analysis Framework for Mission Critical Systems

Author
Rimsha Khan
FALL 2018-MS-18(CSE) 00000278160

MS-18 (CSE)
Supervisor
Dr. Farooque Azam

DEPARTMENT OF COMPUTER & SOFTWARE ENGINEERING
COLLEGE OF ELECTRICAL & MECHANICAL ENGINEERING
NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY
ISLAMABAD

AUG, 2021

Java Code Analysis Framework for Mission Critical Systems

Author

Rimsha Khan

FALL 2018-MS-18(CSE) 00000278160

A thesis submitted in partial fulfillment of the requirements for the degree of

MS Software Engineering

Thesis Supervisor:

Dr. Farooque Azam

Thesis Supervisor's Signature:____

DEPARTMENT OF COMPUTER & SOFTWARE ENGINEERING

COLLEGE OF ELECTRICAL & MECHANICAL ENGINEERING

NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY,

ISLAMABAD

AUG, 2021

DECLARATION

I certify that this research work titled *"Java Code Analysis Framework for Mission Critical Systems"* is my own work under the supervision of Dr. Farooque Azam. This work has not been presented elsewhere for assessment. The material that has been used from other sources has been properly acknowledged / referred.

Signature of Student

Rimsha Khan

FALL 2018-MS-18(CSE) 00000278160

LANGUAGE CORRECTNESS CERTIFICATE

This thesis is free of typing, syntax, semantic, grammatical and spelling mistakes. Thesis is also according to the format given by the University for MS thesis work.

____

Signature of Student

Rimsha Khan

FALL 2018-MS-18(CSE) 00000278160

____

Signature of Supervisor

COPYRIGHT STATEMENT

ACKNOWLEDGEMENTS

I am extremely thankful to Allah Almighty for his bountiful blessings throughout this work. Indeed this would not have been possible without his substantial guidance through each and every step, and for putting me across people who could drive me though this work in a superlative manner. Indeed none be worthy of praise but the Almighty.

I am profusely grateful to my beloved parents for their love, prayers, support and sacrifices for educating and preparing me for my future. I also thank my siblings who encouraged me and prayed for me throughout the time of my research.

I would also like to express my gratitude to my supervisor **Dr. Farooque Azam** and my co-supervisor, **Dr. Muhammad Abbas**, for their constant motivation, patience, enthusiasm, and immense knowledge. Their guidance helped me throughout my research and writing of this thesis. I could not have imagined having a better advisor and mentor for my MS study.

I would like to pay special thanks to **Muhammad Waseem Anwar** for his incredible cooperation and providing help at every phase of this thesis. He has guided me and encouraged me to carry on and has contributed to this thesis with a major impact.

I would also like to thank my Guidance Committee Members **Dr. Wasi Haider Butt** and **Dr. Arsalan Shaukat** for being on my thesis guidance and evaluation committee. Their recommendations are very valued for improvement of the work.

Last but not the least, I would like to express my gratitude to all the individuals who have rendered valuable assistance to my study.

Thanks for all your encouragement!

Dedicated to my beloved parents whose tremendous support and cooperation led me to this wonderful accomplishment.

ABSTRACT

Sometimes a single bug can cause loss of millions of dollars as in the case of Ariane or a small glitch in the software can cause loss of life as in the Therac 25 case where 7 people died due to overdose of radiation. Such problems in software proves the importance of correctness of software code and use of quality assurance practices specially for mission critical software where a fault in the software can lead to high financial loss or even loss of life. Since Mission Critical Systems are real time in nature, therefore mostly run time errors in these systems occur due to Concurrency and Logical Errors. A detailed literature review of 51 research papers on Code Analysis indicates the lack of framework for automated code analysis of Java Code regarding Concurrency and Logical errors. Furthermore, the framework proposed and the industrial tools do not check compliance to NASA's coding standards which is a very important standard guide for MCS. Hence there is a sheer need of developing a Java Code Analysis framework for MCS that checks code adherence to NASA's Coding Standards related to concurrency and logical errors.

Keeping this in view, an open source framework for Java Code Analysis of MCS has been proposed that ensures improved software reliability and early detection and correction of code which is very costly at later stages of SDLC. Our analyzer checks Java code compliance to Coding Standards by automating twelve of NASA's coding standards related to Concurrency and Logical Errors. Concurrency includes API Misuse, Synchronization, Thread Safety and Waiting related rules. The framework uses a hybrid code analysis technique made up of Syntactic Code Analysis and Flow Analysis, making use of the benefits of both i.e. imposing rules based on a context free grammar (CFG) and assessing control flow of the test code. Our framework not only detects the violation of a rule but also pin points the location of the rule violation and suggests a fix for each of problem. We analyzed twelve open source Standard Java Projects using our framework to check the validity of our framework. Furthermore, we also induced 7 projects with rule violation and our framework successfully detects those violations. Based on our results we have created a dataset of Logical and Concurrency errors in MCS

Keywords: Code Analysis, Static Code Analysis, Java, Syntactic Analysis, Flow Analysis, Software Quality, Software Reliability.

TABLE OF CONTENTS

TABLE OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

CHAPTER 1: INTRODUCTION

This section provides a detail introduction about the important concepts related to our research, the current problem and an overview of our solution. It is organized into five sub sections. **Section 1.1** describes the background study, **Section 1.2** provides the problem statement of research, **Section 1.3** discusses the proposed methodology, **Section 1.4** gives the detail about research contribution, and thesis organization is presented in **Section 1.5**

1.1. Background Study

The purpose of this section is to introduce the background study of multiple important concepts which has been used in this research. These concepts include:

- ➢ Mission Critical Systems
    - Types of Critical Systems
    - Importance of Critical Systems
    - Examples of Real Time System
    - Real Time and Mission Critical Systems
- ➢ Code Analysis
    - Types of Code Analysis
    - Importance of Code Analysis
    - Important Code Analysis Techniques
- ➢ Code Analysis Standards

1.1.1. Mission Critical System

Mission Critical System (MCS) is any systems which is critical for an organization to the point that a failure in the system can cause serious damage to the organization. These systems are usually all-encompassing or very deep because of integration with core elements of the business. Examples of Mission Critical System include electricity grid system, online banking system and aircraft control system.

Types of Critical Systems

Following are the three types of critical systems:

1. ***Mission Critical System***
   If this type of system fails it will lead to failure of one or more goal directed activity. A specific example of such a system would be a spacecraft navigation sys.
2. ***Safety Critical System***
   If this type of system fails it may lead to injury to the living, serious damage to the environment or even death. For example a chemical manufacturing plant control system.
3. ***Business Critical System***
   When this type of system fails it may result in very high financial damage to a business. For example a bank's customer accounting system.

Examples of MCS

MCS are endorsed and used by all business companies and organizations if these systems are functioning properly. A malfunctioning filtration system can lead to closure of water Filtration Company. Similarly, if a gas critical system malfunctions a number of restaurants and bakeries would shut down and wait for system to function again. Thus, many other examples can be found around the world where if a critical system fails it has serious implications on industries and organizations.

Aircraft Navigation System

All Airline Companies are highly dependent on its navigation system. Aircrafts require navigation system to aid pilots calculate time and distance utilizing Dead Reckoning. Radio-navigation is especially useful in conditions such as low visibility. GPS can also be utilized to provide precise data on location that is inclusive of speed, position and track.

Nuclear reactor safety system

Nuclear Stations use controlled fission chain reaction to produce energy in order to make electricity. Here medical isotopes can also be generated and various researches can be done. The control system for nuclear reactors is of critical importance as a system failure in this case will be devastating not only for the industry but for the community and country at large. Hence, such a system is of international concern. Sensitive detectors are employed in the reactors to monitor and control any deviations of temperature, power generation, pressure levels at various points and input and output of water.

Importance of Mission Critical Systems

Mission critical systems in today's world have no doubt gained popularity and are being employed in most aspects of human life. From medical to business corporates, factories to marketing, power generation to transport, all employ critical systems one way or the other. Its importance can not only be seen in its widespread usage and dependency of people on it rather its value is ascertained by the fact that it is not affordable, financially and otherwise, to interrupt a system because system failure at any point can lead to severe disruption of services, disruption of production, heavy financial losses and can even endanger human life.

Since there is a high cost of malfunctioning of critical systems, to develop critical systems trusted methods and previously tried techniques are preferably used instead of developing new methods that have not been subjected to practical trials. Hence, when employing a critical system an older system is naturally chosen as its merits and demerits are well understood rather than choosing a system new in the market which appear appealing on the outside but its long term problems are yet to be known.

Real Time and Mission Critical Systems

Mission Critical software are mostly real time in nature. Real Time System is a system which fails if a timing deadline is not met. It can be:

- Hard Real Time System – These are systems which fail for any missed deadline
- Soft Real Time System – These systems are tolerant of missed deadline

1.1.2. Code Analysis

Code Analysis is used to find potential defects in the code such as logical errors, deadlocks, useless code, code clone etc. It is normally done early on in the software development life cycle which helps detect and correct flaws in the software which can become very expensive to maintain in the later stages.

Types of Code Analysis

There are essentially three main types of Code Analysis:

1) Static Code Analysis: This is the analysis type in which code analysis is performed without executing the code. It is usually done manually but recently some automation of static code analysis techniques has also been observed in literature. In this technique code is checked at compile time and not during the runtime. It examines code at early stages of development and testing, therefore it is more of a precautionary measure. Inferences are made about what the problems might arise in code before the execution of code based on the results of static code analysis. Besides this it can also be used to improve the readability and maintainability of the source code

2) Dynamic Code Analysis: This is the code analysis type in which code analysis is done by actually executing the code. Dynamic analysis can be as simple as fixing a bug in code and then running it to see if the error still exists. Performing unit test is also dynamic code analysis. Additionally, it can be used to analyze security related bugs because a code interaction with other system components can also be checked by executing the code.

3) Hybrid Code Analysis: In this type of code analysis one or more of static and dynamic analysis techniques are combined to give form of a hybrid of Static and Dynamic Testing. It has proved to be more effective method of code analysis in many cases [9], [11], and [27].

Static code analysis can prove to be better than dynamic code analysis when it comes to locating a problem identified by analysis. Static analysis pin points the location of fault in the code and hence offers quicker fixes. But its high rate of false negatives gives developers a false sense of accomplishment that all security and quality concerns are being tackled. Dynamic code analysis can help developers detect faults in software that might have been a false negative in case of static code analysis. To tackle all the different types of bugs in a software static analysis should be combined with dynamic analysis i.e. Hybrid Analysis.

Importance of Code Analysis

Sometimes a single bug can cause loss of millions of dollars as in the case of Ariane [52] or a small glitch in the software can cause loss of life as in the Therac 25 [53] case where 7 people died due to overdose of radiation. Such problems in software proves the importance of correctness of software code and quality assurance practices. Source code analysis not only helps in reverse engineering and reengineering of software applications but it also helps in maintenance and optimization of the software by making the software more readable. The software sizes are increasing with the advancement in software development. By the year 2025 the software size will increase up to 1 trillion line of code [54]. This alarming increase in the size of a software and the problems it causes shows the importance of automated ways of code analysis to prevent defects in software and specially Mission Critical Systems.

Important Code Analysis Techniques

The techniques employed for static code analysis range from elementary approaches like pattern based approaches to more complex ones like Syntactic Analysis, Flow Analysis, Taint Analysis and Machine Learning. One or more techniques can be combined to form a **hybrid** which has proven to be more effective [2], [18],[37]. Most widely used technique is the Syntactic Analysis. It is based on imposing rules, implied by a context-free grammar, on syntax tree or program. It can be:

- Top down: In this approach the parse tree starts creating from the top i.e. the root and proceeds towards the bottom i.e. the leaves.
- Bottom- down: In this approach the parse tree is constructed from the leaf and proceeds towards the roots.

Using top-down parsing has advantages over Bottom down approach such as use of more general grammar, easier to debug, and passing values (attributes) both up and down the parse tree.

Flow analysis is yet another technique is commonly used for analysis. It ensures analysis of control or data flow around the system using graphs. In Flow analysis, code is checked for defects by first making a graph structure from the code e.g. data flow graph or control flow graph, which is then traversed to root out problems in a program. A call graph starts from the main method and forwards to all methods recursively through method calls, representing global dependencies in a program. This straight forward approach can be used for analysis in procedural languages where there are no dynamically bound method calls and instances as in object oriented languages. Other techniques include taint analysis i.e. Analysis of variables which can be modified by user and techniques based on Machine learning and formal methods.

1.1.3. Code Analysis Standards

Due to the importance of Code Analysis in system development, several standards have been proposed. A coding standard is a set of best practices for developing software of better quality Standards such as ISO, IEC, MISRA makes software safer. Code analysis makes checking of compliance to coding standards easier.

Java is renowned language for the development of MCS's [62] and few Industry wide accepted Java Coding Standards are given in Table 1.

Table 1.1: Java Coding Standards

Coding Standards	**Publisher**	**Primary Focus**
Google Style Guide	Google	*Hard-and-fast* Coding Style rules
Sun Java Coding Standards	Oracle	Maintainability
JPL Coding Standards	NASA	Run Time Errors for Mission Critical Systems

NASA's JPL Coding Standards

When it comes to mission and safety critical software in Java the best and most practical standards are the JPL Coding standards proposed by NASA. These coding standards addresses potential risk in a software related to multi-threaded software. It is specially designed for ground mission critical systems to improve code quality by minimizing the possibility of run time errors in the code. The standards are based on the MISRA and "Power of Ren" coding rules.

NASA's JPL Coding standards are divided into three main categories based on their criticality level which are:

1) Critical
2) Important
3) Advisory

1.2.Problem Statement

Currently most of the papers have used Code Analysis to target issues related to security, duplication, complexity, readability, memory use, unnecessary code, missing code, etc. Static, Dynamic, or Hybrid Code Analysis techniques are being used to detect problems in the code. Research shows that concurrency and Logical Errors are most important reasons for failure of Real Time Mission Critical Systems. NASA's JPL Standard has rules related to these two areas but they have not been automated by any tool. State-of-the-art indicates the lack of framework for automated Java code analysis regarding logical errors in MCSs. In addition, the industrial code analysis tools do not target Java based MCSs particularly for logical and runtime errors. Code Sonar is a very important tool that checks code compliance to many important standards but it does not check code conformance to NASA's JPL Standards for Java coding language. Enforcing these rules on Mission Critical Systems can greatly mitigate rate of system failure. Hence, there is a dire need for a framework that can analyze Mission critical Systems by automating JPL rules related MCS. Figure 1 shows a summary of problem statement.

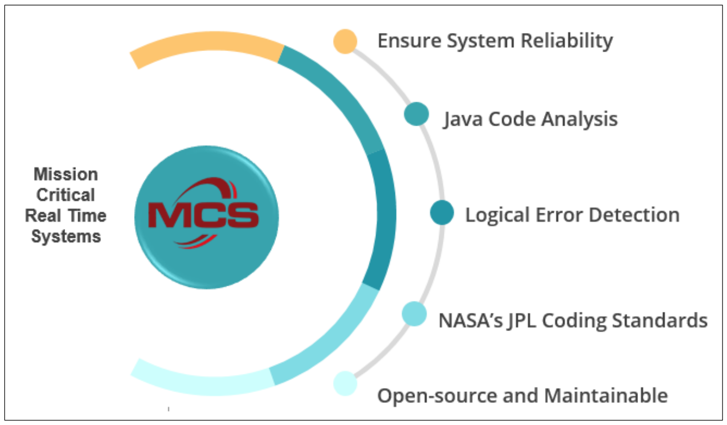

Mission Critical Real Time Systems
MCS
Ensure System Reliability
Java Code Analysis
Logical Error Detection
NASA's JPL Coding Standards
Open-source and Maintainable

Figure 1.1: Problem Statement Summary

1.3.Proposed Methodology

Entire research is done in a very systematic way. **Figure 1.2** represents the flow of research step by step. In first step we identify the problem. Then proposed the ideal solution for the problem identified in first step. We carried out a detailed and comprehensive literature review which helps us to identify the optimal solution for the problem. We reviewed the researches carried out related to our proposed solution, analyzed and compared it. Then we implemented our framework using some tools and techniques. Our proposed framework is then validated using some RTMCS.

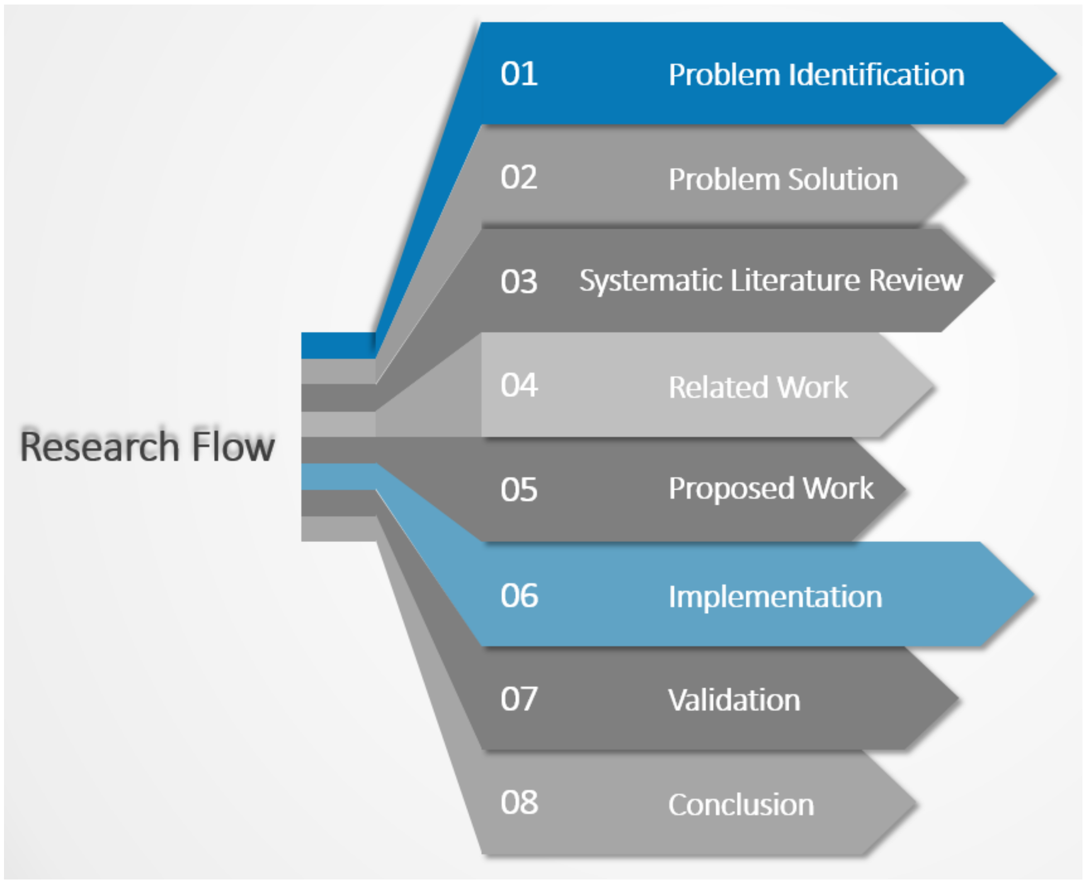


Figure 1.2: Research Flow

1.4.Research Contribution

The contribution of our research is a complete, open source framework for Code Analysis of Mission Critical System by checking code compliance to coding standards proposed by NASA'S JPL Laboratory. Detailed set of contributions of the proposed approach are as follows:

- Improve reliability of Software and Mission Critical Software in particular because of the high criticality factor.
- Automation of NASA's Rules related to logical errors and concurrency as it an important rule set specially designed for Mission Critical Systems.

- Detection of potential errors points and deadlocks in the code that can cause run time errors
- Location of the cause of errors in the code by displaying line number of the problematic code.
- Suggestion on how the run time error can be avoided or fixed based on Standards
- Early detection and correction of code which is very costly at later stages because maintenance cost increases in the later stages of Software Development Lifecycle (SDLC)
- The proposed work has been validated using Mission Critical case studies including Elevator Control System, Autonomous Driving and Aircraft Control System.

1.5.Thesis Organization

Organization of the thesis is represented in **Figure 1.3. CHAPTER 1:** offers a brief introduction containing the background study, problem statement, research contribution and thesis organization. **CHAPTER 2:** provides the detailed literature review highlighting the work done in the domain of Code Analysis on general and Code Analysis of Mission Critical Systems in specifuc. Section one presents a systematic literature review on Code Analysis techniques and tools. Section 2 describes the code analysis review from industrial perspective by presenting a review of all the different code analysis tools available in Market for JAVA Language. **CHAPTER 3:** covers the details of proposed methodology used for identification and solving of the problem inhand. **0** presents the detailed implementation of our framework, architecture along with its interface.Error! Reference source not found. **CHAPTER 5:** provides the validation performed for ur proposed methodology using two important case studies, including Elevator Control system and Bank System. **CHAPTER 6:** contains a brief discussion on the work done and also contains the limitations to our research. **CHAPTER 7:** concludes the research and recommends a future work for the research.

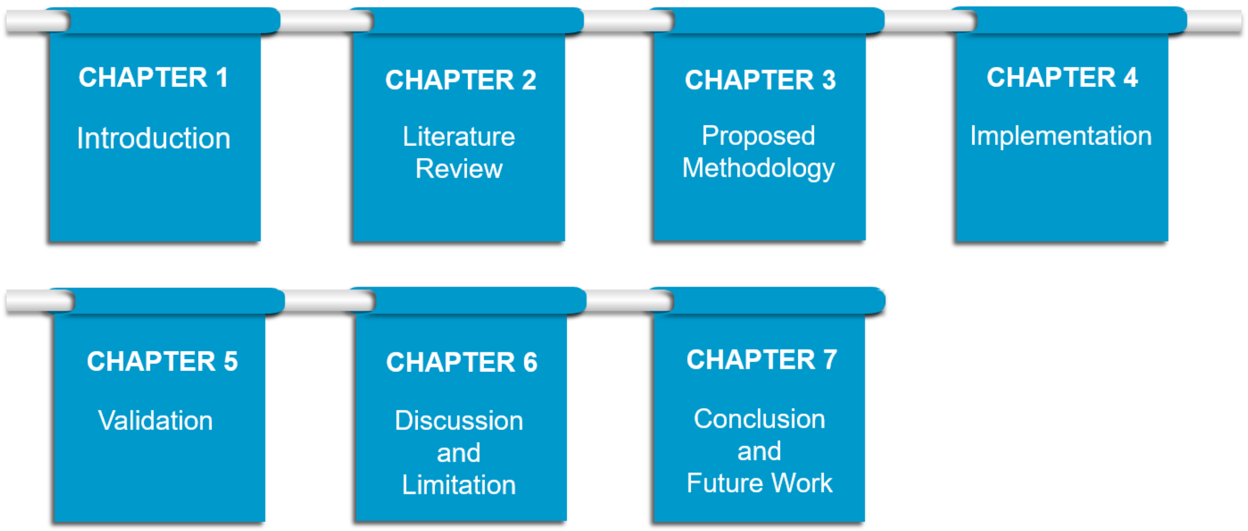


Figure 1.3: Thesis Outline

Chapter 2

Literature Review

CHAPTER 2: LITERATURE REVIEW

Code analysis has been an active field of research for many years. Hence there are a number of review papers related to code analysis and its sub fields i.e. static, dynamic and hybrid code analysis. A survey on static code analysis [59] provides the comparison of 4 best algorithms for static code analysis against mathematical logic language for model checking. Another review paper [60] only focuses on vulnerability detection using static analysis of C/C++ code by comparing the results of 11 different open source tools. Our review is different from existing review papers because it provides a bigger picture of code analysis by presenting static, dynamic and Hybrid code analysis approaches in one place so as to provide an easier way of comparing the three approaches. To the best of our knowledge no other paper provides a latest review of code analysis techniques and tools proposed or implemented between the year 2014 and 2020 that provides an overall comparison between the different approaches of code analysis i.e. static, dynamic and hybrid code analysis. The scope of our study is restricted only to the research studies published between the year 2014 and 2020 that implement, propose or improve a technique or tool of code analysis in the area of static, dynamic and hybrid code analysis. We have further restricted our scope by not including studies related to code analysis that targets code clone detection because a recent review paper on code clone detection [61] covers all its aspects comprehensively.

We analyzed Code Analysis developments from the following perspective:

- The different studies reported in literature for Code analysis in general and MCS's in particular.
- Industrial tools proposed by researchers for Code Analysis

We have carried out a Systematic Literature Review (SLR) on 51 research articles, comprising both conference and journal papers selected from the year 2014- 2020 after a detailed search process.

2.1. Systematic Literature Review

2.1.1. Review Protocol

Two components of the review protocol i.e. Research questions and background of the study have been discussed in the last section i.e. Introduction. This section presents the remaining five important components of the total seven basic components of review protocol.

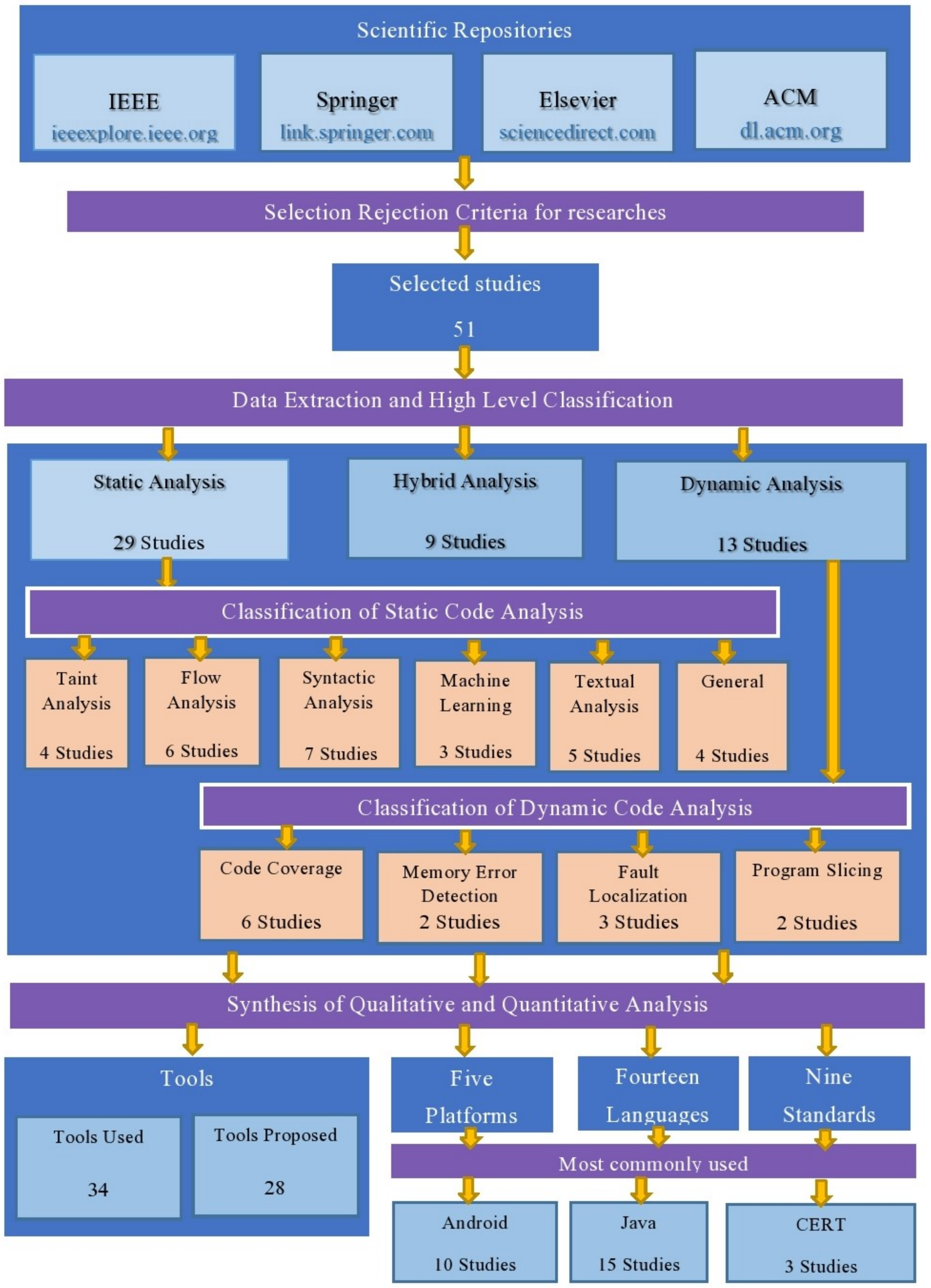


Figure 2.1: Overview of review process

A. Categories of Code Analysis:

For the simplification, we have define three major categories of code analysis. The description of these categories is given below.

1) Static Code Analysis: It is a code analysis technique that checks code at compile time and not during the runtime. It examines code at early stages of development and testing, therefore it is more of a precautionary measure. Inferences are made about what the problems might arise in code before the execution of code based on the results of static code analysis. Besides this it can also be used to improve the readability and maintainability of the source code.

2) Dynamic Code Analysis: It is an analysis technique that checks code by executing it. Dynamic analysis can be as simple as fixing a bug in code and then running it to see if the error still exists. Performing unit test is also dynamic code analysis. Additionally, it can be used to detect security related bugs because a code interaction with other system components can also be checked by executing the code.

3) Hybrid Code Analysis: It is a code analysis technique that formed by the combination of both Static and Dynamic techniques.

B. Selection Rejection Criteria

To achieve the required goals of SLR and effectively answer our research questions some rules were pre-defined based on which research studies were filtered out. Hence a selection and rejection criteria was clearly defined which is given below.

- Only publications that propose a tool or technique for code analysis techniques i.e. Static, Dynamic or Hybrid Code Analysis are included in the study.
- Only the publications from 2014 to 2020 are included in review process and the rest are rejected to ensure a latest study of research available on code analysis.
- The papers were only selected from four well-accepted scientific databases i.e. Springer, Elsevier, ACM and IEEE. Studies published on other repositories are not considered for review.
- In case of two papers with almost similar content were found then only one of them was included in review and the other is discarded.

- The research paper that comprised of 2 to 3 pages and were not full length papers were discarded from the study.
- Research that was published in any language other than English were not included in the study.
- Only the papers published in conference proceedings or journals were included in the study.
- Research papers that perform code analysis to perform code clone detection are excluded from the study because it is thoroughly covered in a recent SLR on code clone detection [61].

C. Search Process

In search process firstly we searched the four databases we have selected (IEEE, ACM, Elsevier and Springer) as described in the criteria for selection and rejection. A summary of the search process is presented in FIGURE. As the paper presents a review on the advancement code analysis techniques in the recent years therefore only the papers that proposed some new technique or tool for static, dynamic or hybrid code analysis were selected.

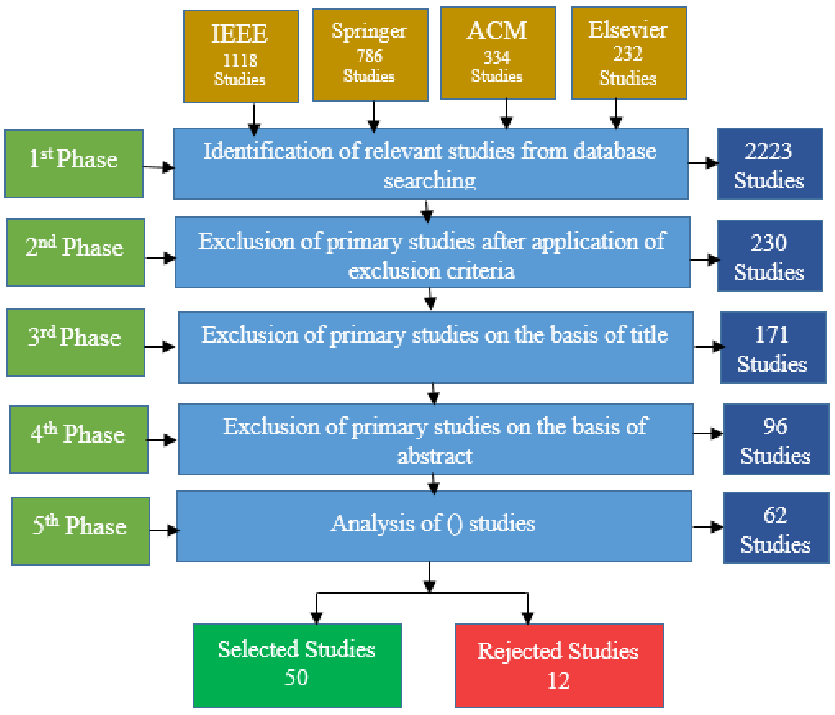


Figure 2.2: Search Process Flow

We followed the steps presented in the figure to select the studies that are most relevant to code analysis and our focus. In first step all the databases are searched for using a variety of keywords along with operators (AND, OR) are used to perform the search process. Using advanced search in the selected databases certain constraints including time frame were imposed on search to get controlled number of results. In the next phase 230 papers are shortlisted from 2223 studies based on other criteria of selection and rejection which are discussed in the previous section i.e. Section II (C). In the third and fourth phase more papers are excluded on the basis of title and abstract leaving 171 and 92 studies respectively. In the 5th phase overall analysis of studies was done by skimming through the papers which further narrowed down the selection to 51studies and rejected 12 studies. The remaining 51 studies at the end of search process completely comply with our selection and rejection criteria.

Table 2.1: Summary of Search Terms and Corresponding Results.

Sr.#	Search Terms	Operator	IEEE	ACM	Springer	Elsevier
1.	Code, Analysis	AND	27000	10021	19221	9238
2.	Static, Code, Analysis	AND	1653	753	547	7543
3.	Dynamic, Code, Analysis	AND	6123	432	234	323
4.	Hybrid, Code, Analysis	AND	3212	121	87	65
5.	Dynamic, Code, Analysis, Code, Coverage	AND	343	5	3	6
		OR	932	327	343	398
6.	Dynamic, Code, Analysis, Fault Localization	AND	237	15	6	12
		OR	1023	276	198	182
7.	Dynamic, Code, Analysis, Memory, Error, Detection	AND	134	5	3	1
		OR	1294	65	323	176
8.	Dynamic, Code, Analysis, Program, Slicing	AND	91	12	11	5
		OR	2311	654	297	132

D. Quality Assessment

To achieve more reliable results, we narrowed our sources of SLR to only the most reliable and popular databases i.e. IEEE, ACM, Elsevier, and Springer. A total of 31 studies are selected from IEEE including 6 journal papers and 25 conference papers. From ACM a total of 8 papers are selected from which 5 are conference papers and 3 are Journals. From Elsevier 5 journal papers and 1 conference paper is selected. From Springer a total of 5 papers are selected which are all conference papers. Table 3 presents a summary of publications selected from each database and the publication type. **Database** table heading represents the scientific repositories name from which the papers are selected. For each paper the citation of the papers is written against the database from which it is taken and under their respective publication type i.e. Journal or Conference. **Total** represents the aggregate of the conferences and journal papers in each scientific repository.

Table 2.2: Summary of research papers based on scientific database and publication type

Database	**Journal papers**	**Conference papers**	**Total**
IEEE	[1][2][3][4][5][7]	[8][10][11][12][15][16][17][19][20][21][22][23][24][25][26][27][28][29][31][32][39][40][41][44][46][51]	32
Elsevier	[33][34][36][37][38]	[45]	6
ACM	[6][9][13]	[42][43][47][48][49]	8
Springer	Nill	[14][18][30][35][50]	5

The overall number of publication w.r.t database and publication type is presented in the form of graph in figure no. Yellow bars in the graph represents the number of journal papers, Blue bars represents conference papers and yellow bar represents journals and conference papers combined.

To assess the most recent advancement in code analysis we have selected the publications only between the year 2014 and 2019. We also found a paper from the year 2020. From year 2019 we selected 5 publications. 9 publications from the year 2018, 11 publications from the year 2017, 13 publications from the year 2016, 5 publications from the year 2015 and 6 publications from the year 2014 are selected for the study. A summary of the publication selected per year is presented in Table 4. The **Year** represents the year of paper that is selected for the review. **Reference** represents the citation number of the reference number of the selected paper in the reference section of this paper. **Total** presents the total number of publications selected in each year.

Table 2.3: Summary of selected publication per year

Year	References	Total
2014	[3][5][6][18][27][48]	6

2015	[8][29][39][40][50]	5
2016	[2][4][12][14][15][17][31][35][36][38][43][45][46]	13
2017	[1][15][19][22][23][24][30][40][44][47][49]	11
2018	[7][10][11][24][26][28][32][33][42]	9
2019	[9][13][20][21][37]	5
2020	[34][51]	1

E. Data Extraction and Synthesis

We extracted data and perform synthesis using a template presented in Table 5. Bibliographic Information of the studies are observed for each selected study. The methodology proposed, implementation details, outcomes nag categorization proposed by each study is observed. Furthermore programming languages, target platform, target uses, Tools used and proposed, and Standards in each study are identified. Finally, a comparative analysis of the major categories of Code analysis i.e. Static Code Analysis, Dynamic Code Analysis and Hybrid Code is presented

Table 2.4: Data Extraction and Synthesis template

Sr#	Description	Detail
1.	Bibliographic Information	The title, authors, publication year and type of publication i.e. conference or journal is observed for each of the selected studies.
2.	Proposed Methodology	The methodology proposed by each of the selected research is observed.
3.	Implementation Details	Technique used to implement each methodology is analyzed
4.	Outcomes	Outcomes of each study is analyzed.
5.	Grouping	The selected studies are grouped into categories and subcategories, the result of which are summarized in Table 9 and Table 10
6.	Investigation of categories	Analysis and further classification of each of the major categories i.e. Static Code Analysis, Dynamic Code Analysis and Hybrid Code Analysis, to find answer to RQ1 are discussed in Section II A, B and C respectively. The analysis results for sub categories i.e. Taint Analysis, Syntactic Analysis, Flow Analysis, Learning, Textual Analysis and General Category of Static Code Analysis are summarized in Table 6-11 respectively. Whereas the analysis results for subcategories of Dynamic Code Analysis i.e. Code Coverage, Memory Error Detection, Fault Localization and Program Slicing is summarized in Table 13-17 respectively
7.	Programming Languages	Programming Languages being analyzed in each of the selected studies are presented in Table 19

8.	Target Platform	Target platform in each of the selected studies are presented in Table 20
9.	Uses	Target Use of Analysis method in each of the studies is presented in Table
10.	Tools	Tools used and proposed in each of the studies are presented in Table 21
11.	Standards	A summary of Standards to which the analysis method is checking compliance to in each of the selected studies is presented in Table 22
12.	Comparative Analysis	A comparative analysis of the major categories of Code analysis i.e. Static Code Analysis, Dynamic Code Analysis and Hybrid Code is presented in Table 23

2.1.2. Classification and Analysis

To answer the research questions mentioned before, a total of 51 papers in static code analysis have been examined out of which 15 are journals and 36 are conferences proceedings. Figure presents the conference journal ratio in the form of a Donot chart. Almost 28% are published as journals and 73% are printed in international conferences. These studies are published in different journals including IEEE Transactions on Information Forensics and Security, IEEE transaction on Reliability, ACM Transactions on Programming Languages and Systems (TOPLAS) etc. Similarly a very wide range of conferences are included for study. All these studies have been divided into two major categories Static Code Analysis and Dynamic Code Analysis which are then further categorized into its subcategories.

Table 2.5: Summary Classification result of selected studies

Technique	Definition	References	Total
Static	Code Analysis Technique that checks code without executing it.	[1][2][3][5][7][8][10][12][14][15][16][18][19][20][21][22][24][25][26][29][30][31][32][33][34][36][37][38][51]	29
Dynamic	Code Analysis Technique that checks code by executing it.	[13][35][39][40][41][42][43][45][46][47][48][49][50]	13
Hybrid	Combination of both Static and Dynamic techniques.	[4][6][9][11][17][23][27][28][44]	9

A. Static Code Analysis Classification:

First category of code analysis is the static code analysis which has been discussed in detail in Section 2 of this article. Static code analysis has been further divided into subcategories based on approach of Static Analysis, for ease and clarity in study. The categories are Taint Analysis, Syntactic Analysis, Flow Analysis, Machine Learning, Textual Analysis and a General Category.

Table 7 presents a definition and related papers of each categories. **Category** shows the name of category. **Definition** presents a brief description of that category. **Reference** shows the citation number of the related paper in references of this paper. **Total** presents the total number of papers in each category.

Table 2.6: Summary Classification result of selected studies.

Categories	Definition	References	Total
Taint Analysis	Analysis of variables which can be modified by user.	[14][19][25][33]	4
Syntactic Analysis	Analysis based imposing rules on syntax tree or parsing.	[16][21][22][24][26][29] [31]	7
Flow Analysis	Analysis of control or data flow around the system using graphs.	[1][7][8][12][30][36]	6
Machine Learning	Automatically learning Analysis from experience and data.	[10][32][34]	3
Textual Analysis	Analyzing code using textual properties like stemming, lemmatization, and spell checking	[3][5][15][20][38]	5
General	A combination of one or more of the above approaches.	[2] [18][37][51]	4

a) Taint Analysis

Taint Analysis focuses on any vulnerabilities in code specially caused by injection of some untrustworthy code. It checks the complete flow of information from input to the possible areas that can be affected by malicious inputs to a software. It helps in identification as well as location of vulnerable parts in a source code. Owing to its numerous applications in vulnerability detection it widely used for software security. Table 8 presents a summary of analysis on the research work done on taint analysis, against certain parameters such as technique, case study and accuracy. These terms are predefined below: 1) **Scope** is whether the scope of paper under study is limited to detection only or correction or both. Another possible value of scope can be Aiding static analysis if the proposed technique improves static analysis in some way 2) **Technique** is the specific methodology or algorithm used in the paper for the analysis of code using textual analysis approach. 3) **Case Study** is the dataset using which the referenced paper has been validated through experiments. 4) **Accuracy** is the results of software after validation experiments. Case

Study is further divided in two parts i.e. **Name** is the name of dataset mentioned in the study and **Availability** is whether the dataset is public or private.

Z. Chengyu et al. [14] presents GreatEalton, an extended form of FlowDroid, to detect ransomeware in android. It detects malicious inputs by tracking information between InputStream, its related classes and Cipher objects that encrypts them. Another more common way to perform taint analysis is to first parse the code and form a tree like structure like Abstract Syntax tree (AST) [33] or concrete syntax tree (CST) [19], then perform taint propagation and analysis on the tree. A. Costin et al. [19] performs taint analysis on Lua code by using a summarized list of tainted inputs and sensitive sinks related to Lua code. The SAST tool presented by the study targets web vulnerabilities. Kurniawan et al [33] uses a PHP parser having 140 grammar combinations in the form of AST. These combinations are traversed to detect a tainted flow pattern. X. Yan et al. [25] also detects taint style vulnerabilities in code but it also introduces detection of a new type of taint style vulnerability i.e. function calling control vulnerability.

Table 2.7: Summary of studies using taint analysis approach to perform Static Code Analysis

Reference	**Scope**	**Technique**	**Dataset**		**Accuracy**
			Name	**Availability**	
[14]	Detection	API misuse detection	Contagio Mobile, Virus Total	Public	99%
[19]	Detection	Concrete Syntax Tree (CST).	N/A	Private	N/A
[25]	Detection	Sink Analysis	N/A	N/A	N/A
[33]	Detection	Pattern Recognition	Stivalet	Public	N/A

b) Syntactic / Symbolic Execution Analysis

Static analysis includes inspecting a program elements, its structure and/or by estimating its possible states. Examining the elements of a program can help identify many important issues in a source code. The analysis includes traversing the Abstract Syntax tree AST and checking the nodes that are visited against some predefined rules. Paper [29] presents compliance to rules by mapping between patches to perform code reviews. Most commercially available tools for static analysis e.g. PMD[55] and Findbugs [56] also follow the same procedure. R. Ramler et al [31], in their work extend the ruleset of PMD by implementing 43 more rules. These rules are mostly related to improper use of xUnit framework and maintenance issues. G. Horváth [24], extends Clang compiler, which is an open source compiler that performs static analysis using symbolic execution, to include Cross Translation Unit (CTU) Analysis. This method is used to detect many different errors in a program that span across translation units. T. T. Nguyen et al [21] combines two tools, Rosecheckers and Frama-C/WP, improves code analysis C source code verification and reduce false positives in static analysis results.

The study [16] presents a tool for detecting code smells by extracting SQL queries in Java code, converting it to AST and then running Antipatterns based smell detectors on it. Uninitialized vulnerability more commonly exist in C, C++ and are sometimes difficult to detect. Z. Xu [26] proposes STACKEEPER that detects such vulnerabilities in code at byte-level. The validity of the model is checked on XNU source code. . LibLoader [22] detects missing libraries using Understand, which uses code analysis by comparing code with REST-API of Maven 2. B Shastry [30] presents Orthrus which detects vulnerabilities by constructing an input dictionary based on program and data flow. Table 9 presents a summary of analysis on the research work done on taint analysis, against certain parameters such as technique, case study and accuracy.

Table 2.8: Summary of studies using syntactic/symbolic analysis approach to perform Static Code Analysis

Reference	**Scope**	**Technique**	**Case Study**	
			Name	**Availability**
[16]	Detection	Query Anti-patterns	N/A	N/A
[21]	Detection	Hybrid System	ISOBUS protocol library	Private
[22]	Both	Dependency resolution	Public	Public
[24]	Detection	Exploded graph and inline analysis	Large Industrial projects	Public
[26]	Detection	AST, Uninitialized memory use	XNU	public
[29]	Aid Analysis	Mapping between patches	Eclipse CDT, Eclipse JDT	Public
[31]	Detection	Mapping with Rules	JFreeChart	Public

c) Flow Analysis

Flow analysis are intra procedural techniques having its origins in compiler construction context. In Flow analysis, code is checked for defects by first making a graph structure from the code e.g. data flow graph or control flow graph, which is then traversed to root out problems in a program.. This straight forward approach can be used for analysis in procedural languages where there are no dynamically bound method calls and instances as in object oriented languages. Study [1] makes use of the aforementioned benefits of CFG for procedural languages and combines it with Pattern matching to perform analysis on large scale industrial applications. Author proposes flow analysis based static analysis specifically for android platform by exploiting implicit method invocation processing. The proposed framework finds loopholes in android source code using information flow analysis.

Generally flow analysis is done against a set of rules. Y. Takhma [8] proposes code analysis based on code compliance to standards. In this case an abstract model of source code is first created which is then traversed against a set of XML rules, defined by MyIC phone platform coding standards, to find potential non-compliances to the standards. In study [7] the author proposes a model in which the source code is reduced before mining task by reducing the CFG created and extracting only the relevant portion for analysis, as a result reducing the mining effort and computation time. In another study [36] wide approximations done at joints in a flow, where two control paths meet, is addressed by presenting a generic abstract based precision framework. This methodology improves the precision of analysis done at joint points in a flow analysis.

Table 2.9: Summary of studies using Flow Analysis approach to perform Static Code Analysis

Reference	Scope	Technique	Graph	Case Study	
				Name	**Availability**
[1]	Both	Pattern-matching	CFG, DFG	Injection moulding machines	Private
[7]	Detection	Pre-Condition Mining, CFG Reduction	CFG	Boa datasets, DaCapo and SourceForge	Public
[8]	Detection	Rules Matching	CFG	MyIC Phone Application	Private
[12]	Detection	Method Invocation, privilege analysis	DFG	IMDeveloper,android_auto_sendsms, myAppWeixin	Public
[30]	Detection	Flow Graph, Extended Fuz	CFG	nDPI and tcpdump	Public
[36]	Detection	Predicate Analysis	CFG, DFG	Scade	Private

d) Machine Learning

ML is essentially utilized to examine source code by performing better pattern recognition and identification of violation of some rules. The present paper [10] presents a novel approach of Code Analysis in which Machine Learning is used to recognize patterns in more complex and large software that becomes increasingly difficult to be comprehended by humans. Study [32] uses a new methodology called Software Assurance Personal Identifier (SAPI) to classify results of static code analysis as true or false positive vulnerabilities. It uses probability method and assigns a personal identifier, which is an additional feature and contains information like author name, base on which the results are classified. A. Muhammad [34] propose a malware detector which uses customized learning models concluding that Bidirectional long short-term memory (BiLSTMs) is used to identify the static behavior of Android malware beating the state-of-the-art models without using handcrafted features. Table 11 presents a summary of analysis on the research work done on taint analysis, against certain parameters such as technique, case study and accuracy.

Table 2.10: Summary of studies using Machine Learning approach to perform Static Code Analysis

Reference	**Scope**	**Technique**	**Case Study**		**Accuracy**
			Name	**Privacy**	
[10]	Both	Pattern Recognition	ALLEGRO	Private + Public	N/A
[32]	Aid Static Analysis	Probability using personal identifier	N/A	Private	89.00%
[34]	Detection	Deep Neural Networks	DREBIN, Android Malware Dataset (AMD), VirusShare	Public	99.90%

e) Textual Analysis

Static textual analysis techniques treats code as raw text to perform code Analysis. In this context a tool called STAC [15] is proposed. STAC is a code analysis tool for Java, C++, and C# programming langauges that provides solution for code indexing and process textual patterns inside the code.. S. A. Musavi [3] uses a simple technique of code analysis to detect malicious drivers in the system. A. Bartel [5] and his co-authors have designed a framework as Soot Plugin. It uses String analysis to analyze permission checks in Android. In the study [38] a static code analysis framework is presented, which uses a four layered architecture to check malicious permissions and other dangerous intensions. R. Haas [20] uses code analysis to detect unnecessary code. Table 12 presents a summary of analysis on the research work done on taint analysis, against certain parameters such as technique, case study and accuracy.

Table 2.11: Summary of studies using textual analysis approach to perform Static Code Analysis

Referen ce	**Scope**	**Technique**	**Dataset**	**Accuracy**
[3]	Detection	Feature based analysis	Public	98.15%
[5]	Detection	Class-hierarchy and field-sensitive permission check	Public	N/A
[15]	Detection	Text Extraction, Splitting, and Processing	N/A	98%
[20]	Detection	Stability and centrality of code	N/A	64%-100%
[38]	Detection	Threat-degree threshold model	Private	98.80%

General:

General is the category of static code analysis that combines two more static analysis techniques to devise a hybrid technique and give better results. I. Medeiros [2] presents static analysis technique by combining two apparently orthogonal approaches: taint analysis which includes human coding knowledge about vulnerabilities and datamining which is automatically obtains that knowledge with machine. This type of detection also offers automatic code correction. W Niu [37] presents a method in which static taint analysis id used to find taint propagation paths Bidirectional Long Term Short Term Memory (BLTSM) is applied over it to find vulnerabilities. The proposed system is validated on Code Gadget and NIST dataset achieving an accuracy of 97%. S. A. Mokhov [18] uses classical NLP techniques for detection and classification of vulnerabilities in the code as well as bad coding practices. Authors in [51] modify Java compiler and in included the functionality of computing seven syntactic and semantic representation in for of different graphs like Abstract Syntax Tree (AST), Control Flow Graph (CFG), Call Graph, Type Graph, Program Dependency Graph (PDG), Control Dependency Graph (CDG) and Package Graph using their ProgQuery platform.

Table 13 presents a summary of analysis on the research work done on taint analysis, against certain parameters such as technique, case study and accuracy. These terms are predefined below: 1) **Scope** is whether the scope of paper under study is limited to detection only or correction or both. Another possible value of scope can be Aiding static analysis if the proposed technique improves static analysis in some way 2) **Technique** is the specific methodology or algorithm used in the paper for the analysis of code using textual analysis approach. 3) **Case Study** is the dataset using which the referenced paper has been validated through experiments. 4) **Accuracy** is the results of software after validation experiments. Case Study is further divided in two parts i.e. **Name** is the name of dataset mentioned in the study and **Availability** is whether the dataset is public or private.

Table 2.12: Summary of studies using a combination of two or more approaches to perform Static Code Analysis

Reference	**Scope**	**Hybrid**	**Technique**	**Dataset**		**Accuracy**
				Name	**Availability**	
[2]	Both	ML and taint	SVM	Tikiwik, PhpMyAdmin, etc.	Public	92.60%
[18]	Detection	ML and Textual Analysis	n-grams, NLP and statistical smoothening	NIST	Public	N/A
[37]	Detection	ML & taint	Deep Learning	Code Gadget, (NVD),NIST	Public	97.00%
[51]	Detection	Syntactic, Semantic and Flow Analysis	AST, CFG, Call Graph, Type Graph, PDG, CDG, Package Graph	CUP research group	Public	N/A

B. Dynamic Code Analysis Classification

Next major category of code analysis is the Dynamic code analysis. In this technique, the code is analyzed without running the code. This type of code analysis has been discussed in detail in Section I of this article. For clarity and ease in study Dynamic Code Analysis has also been categorized into further four sub categories based on the type of Dynamic Code Analysis being performed. The categories are, Code Coverage, Memory Error Detection, Fault Localization and Program Slicing. Table 14 presents a definition and related papers of each categories. **Category** shows the name of category. **Definition** presents a brief description of that category. **Reference** shows the citation number of the related paper in references of this paper. **Total** presents the total number of papers in each category.

Table 2.13: Classification result of studies using Dynamic Code Analysis

Type	**Definition**	**References**	**Total**
Code Coverage	Computing the code coverage according to a test suite or a workload.	[13][35][39][40][41][50]	6
Memory Error Detection	Detection of bugs that may cause memory errors such as memory leaks	[42][43]	2
Fault Localization	Locating the buggy code according to failing and passing test cases.	[47][48][49]	3
Program Slicing	The technique of reducing a program to its minimum form such that it still performs its required behavior.	[45][46]	2

a) *Code Coverage*

Commonly high code coverage can be achieved random testing. In random testing a stream on random inputs are generated against which a system is checked. Property based Random testing techniques analyzes the behavior of a system by testing executable predicates on multiple randomly generated inputs. Property based random testing has some drawbacks which are covered by coverage guided property based testing as proposed by [13]. This technique is based on coverage guided fuzzing. A. Sakti et al [40] propose a novel test data generation technique based on searching, which works well in achieving high code coverage in unit class testing. M. K. Alzaylaee [41] proposes a new hybrid system is implemented by combining a random based tool with a state based tool (DroidBot) to increase code coverage and uncover more malicious behaviors.

Traditionally HTML based URL crawlers fails to analyze large parts of novel application which have JavaScript at its core. G. Pellegrino [50] proposes dynamic analysis based way of exploring and analyzing web applications which is implemented in the tool j¨Ak, a web application scanner.

Various runtime verification tools for JVM depend on AspectJ and other aspect oriented programming languages. AspectJ, however enforces some limitations on verification tools e.g. inability to weave Java and Android class libraries. O. Javed [35] proposes a domain specific language DiSL that overcomes the above limitation by featuring an extensible joint point model to avoid restricted joint point model in AspectJ. Another approach for analysis of android application developed in Java language is given by C. Huang [39]. In this approach apk files are decompiled to assembly language to which they insert measurement code, recompile it, repackage it and use the patched binary file to check the rate of code coverage.

Table 15 presents a summary of analysis on the research work done on Code Coverage approach of dynamic code analysis against certain parameters such as scope technique and case study. These terms are predefined below: 1) **Scope** is whether the scope of paper under study is limited to detection only or correction or both. Another possible value of scope can be Aiding static analysis if the proposed technique improves static analysis in some way 2) **Technique** is the specific methodology or algorithm used in the paper for the analysis of code using textual analysis approach. 3) **Case Study** is the dataset using which the referenced paper has been validated through experiments. Case Study is further divided in two parts i.e. **Name** is the name of dataset mentioned in the study and **Availability** is whether the dataset is public or private.

Table 2.14: Summary of studies using Code Coverage approach to perform Dynamic Code Analysis

Reference	**Scope**	**Technique**	**Case Study**	
			Name	**Availability**
[13]	Detection	property based testing	two Coq developments	Private
[35]	Detection	Compiler Construction	DaCapo10 and Scala benchmark suites	Public
[39]	Detection	Instrumentation Code	90 Applications from Google Play	Public
[40]	Detection	Unit testing	Joda-Time, Barbecue, Commons-lang, Lucene	Public
[41]	Detection	Hybrid input generation	McAfee Labs	Public
[50]	Detection	navigation graph	WP, Gallery, Joomla etc.	Public

b) *Memory Error Detection*

Programming languages such as C and C++ have weak/static type systems and are therefore more vulnerable to bugs related to memory misuse at runtime, e.g. type confusion, user-after-free and object bound overflow. These errors causes many security and behavior errors in programs which are developed using these languages. G. Duck et al. [42] presents the use of dynamically typed C/C++, which aims to detect such errors by dynamically checking the "effective type" of each object before use at runtime. This concept is implemented in the form of a tool named EffectiveSan or EffectiveType Sanitizer. Because of this vulnerability an attacker can corrupt programmer intended pointer semantics of a downcasted pointer in a way that is type-unsafe. It is called type confusions and is addressed by both the papers in this category. Study [43] presents TypeSan which ensures efficient performance and minimum memory overhead by using a technique called compact memory shadowing for optimum meta data storage service. Table 16 presents a summary of analysis on the research work done on Memory Error Detection approach of dynamic code analysis against certain parameters such as scope technique and case study.

Table 3.15: Summary of studies using Memory Error Detection approach to perform Dynamic Code Analysis

Reference	**Scope**	**Technique**	**Case Study**	
			Name	**Availability**
[42]	Detection	Dynamic type checking	Annotated LLVM	Public
[43]	Detection	Metadata and type management	TypeSan Test Suite	Public

c) *Fault Localization*

Given a set of tests results, the localization of software faults or the identification of erroneous parts of a program is called fault localization. Most fault localization methods depend on identifying suspicious code chunks by detecting a series of the test case execution pass/fail results. The paper [48] proposes mutation-based fault localization technique. To overcome the deficiencies resulting from relying purely on SBDFL formulae, Genetic Programming (GP) and linear rank Support Vector Machines (SVMs) used for ordering coding chunks based on their chances of having fault. [47]and [49]. FLUCCS is the implementation of approach proposed in [49] which is extended in FLUCCite [47] by including a ternary conditional operator. Table 17 presents a summary of analysis on the research work done on Fault Localization approach of dynamic code analysis against certain parameters such as scope technique and case study.

Table 2.16: Summary of studies using Fault Localization approach to perform Dynamic Code Analysis

Reference	**Scope**	**Technique**	**Case Study**	
			Name	**Availability**
[47]	Detection	Emperical Eualvation	Defects4j 0.2.0 repository, Defects Repository	Public
[48]	Detection	Mutaion Testing	Siemens suite	Public
[49]	Detection	GP, SVMs	Defects4J repository	Public

d) Program Slicing

The technique of reducing a program to its minimum form such that it still performs its required behavior. Dynamic slicing is used during debugging and testing as it can be used for test data generation [45]. M Y Hong et al. introduces program slicing to improve the efficiency of automatic test data generation. The algorithm includes slicing of interest point variables and get the current value from it, then in the branch function, using method of minimization, guide the adjustment of program input. Dynamic slicing is also used for debugging. A. Treffer [46] presents Slice Navigator which makes use of dynamic slicing along with back in time debugging to debug Java Code. It provides features such as summary of relevant program state to assist developers, alternate breakpoints to track last-change and direct reconfiguration of slices.

Table 18 presents a summary of analysis on the research work done on Program Slicing approach of dynamic code analysis against certain parameters such as scope technique and case study.

Table 2.17: Summary of studies using Program Slicing approach to perform Dynamic Code Analysis

Reference	Scope	Technique	Case Study	
			Name	**Availability**
[45]	Detection	Test data generation	N/A	N/A
[46]	Detection	Breakpoints	Open Source business process Engine	Public

C. Hybrid Code Analysis:

Hybrid Analysis is the combination of both static and dynamic analysis techniques. Both methods individually have their advantages and disadvantage. For instance static analysis comes with scalability at the expense of low precision. On the other hand, dynamic analysis has scalability issues while giving high precision. The combination of both these methods can allow side-stepping the shortcomings and multiplying advantages of the individual approaches. Table 19 presents the specific approach of static and dynamic analysis which combine to form hybrid code analysis for each paper that falls in the category of hybrid code analysis. Reference is the citation of the paper under study. For static code analysis the categories are Taint Analysis, Syntactic Analysis, Flow Analysis, Machine Learning and Textual Analysis which are discussed in detail in the previous sections. Similarly the categories for Dynamic code Analysis are Code Coverage, Memory Error Detection, Fault Localization and Program Slicing.

B. M. Padmanabhuni et.al [4] combines static analysis with dynamic analysis to audit Binary Overflows (BOFs). First using test data generated using some simple rules and dynamic analysis some of the vulnerabilities are confirmed. Then dynamic code analysis done by mining static code attributes. O. Tripp et al. [6] present a hybrid security analysis approach for JavaScript program analysis in which the static component performs static string analysis on partially evaluated programs of JavaScript and its frequently accessed DOM functions while the dynamic component performs concretization in dynamic way to maximize coverage. M. Thakur [9] presents a two-step analysis framework called the Precise Yet Efficient (PYE) which includes static analysis and dynamic compilation. The framework helps generate low cost precise results at runtime.

K. P. Subedi et al. [11] propose a tool the CRSTATIC (Crypt-Ransomware STATIC) to identify ransomware families using datamining technique (static analysis) and run time analysis. C. Zhang et al. [17] proposes JD slicer is a dynamic slicer which integrates static analysis with dynamic analysis to assist debugging process of JavaScript code by precisely capturing different types of dependencies including data dependencies, DOM dependencies and control dependencies. A. Gerasimov et al. [23] presents an approach for confirmation of reachability of source-sink defects that are found by static analysis with help of directed dynamic analysis. S. Zhao et al. [27] presents malware detection approach based on extended attack tree (static) and force execution according to runtime behaviors for high coverage (dynamic). DexLego is presented in the study [28] which aids static analysis process by reassembling bytecode data and performing just in time collection. Dead code is detected in [44] using dynamic program slicing for which the input test cases are generated using symbolic execution (static).

Table 2.18: Summary of studies on Dynamic Code Analysis

Referen ce	Static Code Analysis					Dynamic Code Analysis			
	Taint	Textual	ML	Syntactic	Flow	Coverage	Memory	PS	FL
[4]	x	x	✓	x	x	x	✓	x	x
[6]	x	✓	x	x	x	✓	x	x	x
[9]	x	x	✓	x	✓	✓	x	x	x
[11]	x	x	✓	x	x	✓	x	x	x
[17]	x	x	x	x	✓	x	x	✓	x
[23]	x	x	x	✓	x	x	✓	x	x
[27]	x	x	x	✓	x	✓	x	x	x
[28]	x	x	x	✓	x	✓	x	x	x
[44]	x	x	x	✓	x	x	x	✓	x

2.1.3. Analysis Results

The selected papers were analyzed w.r.t its programming languages, Target Platform, Target Use, Tools, Standards and Finally a comparative Analysis of the three major code analysis categories, i.e. Static Code Analysis, Dynamic Code Analysis and Hybrid Code Analysis. The result of analysis of each of the given aspect is discussed in detail in the subsequent sub-sections.

A. Programming Languages

A multitude of programming languages have been analyzed in different papers under study. A summary of Programming/Scripting/Query Languages and the papers that are targeting it is presented in table 20. This table can be beneficial to researcher and programmers targeting code analysis tool development to see which programming language have been targeted by researchers more frequently and which languages have need of further research. According to our study, most of the researchers working on code analysis have targeted Java Programming Language [5][7][9][10][12][14][16][18][22][27][29][31][32][40][41][46][47][49][51]. One major reason for this is its widespread use in different types of developments domains including Web, Desktop and most commonly Android application development. C and C++ are the second and third, respectively, most frequently targeted programming languages for code analysis which researchers are working on recently. C is important because of its worldwide use in a multitude of applications; most commonly operating systems and advanced scientific systems. C remains particularly popular in the world of operating systems for example Linux Kernel. Therefore it is targeted by many researchers [4][18][21][23][24][26][36][37][42][44][48]. C is a procedural language, which means that the programmers has to give step wise instruction to the CPU. A straight forward approach such as call graphs can be used for analysis in procedural languages where there are no dynamically bound method calls and instances but it becomes a little more challenging in object oriented languages, such as C++ and Java. Different researchers come with different solutions for it [33][40] etc. C++ and is the main language for enterprise-class, networked applications, therefore various different tools and techniques have been proposed for its analysis [4][15][18][24][26][37][42][43]. C# is another important language from the C family and is quite similar to Java. Analysis of C# code has been studied in [15][24]. JavaScript [6][17][50] and PHP [2][25][33] are two important web development programming languages for client and server side programming and are vulnerable to many security attacks. Markup Languages like XML and HTML are used for front end structuring of android and websites respectively. Sequential Function Chart (SFC) is a procedural language and is analyzed using Flow Analysis approach [1]. Many researchers have also worked on code analysis of domain specific languages such as Lua [19], Boa [7] and DiSL [35]. In some studies [20] and [30] some researchers have only proposed a general technique or tool for code analysis without mentioning any specific programming language.

Table 2.19: Analysis result of selected studies w.r.t Programming/Scripting/Query Language

Programming/Scripting/Query Language	**Reference**
Java	[5][7][9][10][12][14][16][18][22][27][29][31][32][40][41][46][47][49][51]

C	[4][18][21][23][24][26][36][37][42][44][48]
C++	[4][15][18][24][26][37][42][43]
C#	[15][24]
Bytecode	[11][28][34]
JavaScript	[6][17][50]
PHP	[2][25][33]
XML	[8][39]
HTML	[50]
SQL	[16]
Functional Programming Language	[13]
Sequential Function Chart (SFC)	[1]
Assembly	[11]
Lua	[19]
Boa	[7]
DiSL	[35]
General	[20][30]

B. Type of System

The selected papers were analyzed based on the type of system. Based on our focus of our study we classified target systems into three main categories. Mission Critical, Traditional and Other Systems. Mission Critical Systems are systems whose failure can be fatal for an organization. 14 papers proposed a code analysis technique for mission critical including [7], [9], [10], [14],[18], [26], [28], [30] [31], [35],[37],[38], [45] and [50]. Traditional system are system that are not critical to organization. The studies [1], [2], [4], [8], [11], [13], [15], [16], [19], [20], [25], [32], [33], [34], [36], [40], [41], [42], [49], [51] focused on such systems. Whereas the studies, [3], [5], [6], [12], [17], [21], [22], [23], [24], [27], [29], [39], [43], [44], [46], [47], [48] proposed code analysis framework for multi-purpose systems.

Table 2.20. Classification of studies w.r.t. System Type

System Type	**Definition**	**References**	**Total**

Mission Critical	Code Analysis Technique Particularly for MCS	[7], [9], [10], [14],[18], [26], [28], [30] [31], [35],[37],[38], [45], [50]	14
Traditional	Code Analysis Technique for traditional	[1], [2], [4], [8], [11], [13], [15], [16], [19], [20], [25], [32], [33], [34], [36], [40], [41], [42], [49], [51]	20
Other	Multi-purpose	[3], [5], [6], [12], [17], [21], [22], [23], [24], [27], [29], [39], [43], [44], [46], [47], [48]	17

C. Mission Critical Systems

14 out of 50 studies were identified to be targeting MCS and only 5 tools out of these 14 tools perform code analysis of Java language. Since our research is focused on Mission Critical in Java, we further analyzed studies related to MCS in Java. The summary of analysis is presented in table 22. Reference represents the citation number of the references provided in the reference section of this thesis. Technique means the code analysis technique used. Tool means the name of tool or framework proposed in the study. Some studies have not given their framework a name so we have written N/A in place of it. Standard represent the name of Coding standards the the framework check compliance to if any. Error means the type of target error type of the framework/tool.

Table 2.21: Summary of Mission Critical Systems

References	Technique	Tool	Standard	Error
[14]	Data Flow Analysis	GreatEatlon	N/A	API misuse detection
[15]	Textual Analysis	STAC	N/A	Indexing, Spell Check
[22]	Dependency resolution	LibLoader Application	N/A	Missing Code
[31]	Mapping with Rules	N/A	Google Coding Standards	Assert, Naming Conventions, Setup Teardown routines
[49]	Empirical Evaluation	FLUCCS	N/A	Fault Localization

C. Standards

Some studies [2][9] are guided by the OWASP (Open Web Application Security Project). It is a non-profit organization that assists companies to create, purchase, and maintain trustworthy software applications by educating its employees related to software, about common Web

Application security vulnerabilities and its risks. The popular Top Ten Web Application Security List is also published by OWASP. The Cheat Sheet evidently spreads out the key prerequisites to anticipating every defenselessness top to bottom, and over that incorporates testing agendas and advisers to a guarantee your code's veracity. CERT is a secure coding standard available for C and C++ [21], [30]. CERT also has coding standards for Java which includes 83 recommendations out of which 13 are implemented as static analysis. It was designed specifically to be enforced by software code analyzers using static analysis techniques. What's more, ISO/IEC TS 17961 helps to ensure fewer false positives are identified when a static code analyzer is used. Google provides some style guidelines that can be used as conventions for naming etc. to improve code readability and maintainability. SQL Antipatterns is a book by Bill Karwin, discussed in [16], shows all the common mistakes of database programming, how to avoid those pitfalls and what the best fixes for those problems are. Table 23 presents analysis results of selected papers with respect to compliance to a standard.

Table 2.22. Analysis result of selected studies w.r.t compliance to a standard

Standards	Reference
CERT	[21][30][51]
ISO-IEC	[1][42]
Google Coding Standards	[29][31]
OWASP Top 10	[2][19]
Unix ACL	[5]
MyIC phone platform coding standards	[8]
SQL Antipattern by Bill Karwin	[16]
Standard template library (STL),	[43]
WIVET	[50]

2.2. Industrial Perspective

Reading code to find defects can be very difficult or even impossible with the ever increasing size of code. Therefore several tools are available in the market that can help analyzer detect problems in the code early in the SDLC. These tools can be used to detect different types of problems in the code like improper naming conventions and code clones. Some of these errors can be fetal for systems and MCS in particular. We analyzed different code analysis tools and selected 8 best tools for Analysis of code in Java language. Table 24 presents a summary of tools for Java language. **Tools** represent the name of code analysis tool for Java. **Availability** is whether the tool is available open source or not. **Standard** represents the Coding standard the tool checks compliance to. **Input** is the type of input the tools accepts. It is normally in the form of source code or byte code. **Error** is the type of errors that are detected by the tool e.g. concurrency errors, code clone etc. **Output** is the form in which output is presented by the tool. **MCS** means whether the tool is built for Mission Critical System or not. **Extensibility** means whether the tool can be extended or not.

Table 2.23: Summary of Industrial Tools for Code Analysis of Java

Sr #	**Tool**	**References**	**Availability**	**Standard**	**Input**	**Errors**	**Output**	**Extensibility**
1	Checkstyle	[79]	Open source	Google Java Style	Source code	Style conventions.	List, XML, HTML	Possible
2	DCD (Dead Code Detector)	[80]	Not Open Source	N/A	Bytecode	Detect dead code	List	Not Possible
3	Dependency Finder	[81]	Open source	N/A	Bytecode	Dependency check	Graph	Possible
4	FindBugs	[56]	Open Source	Sun Standards	Source code, byte code	Potential bugs and performance issues	List, XML	Possible
5	JLint	[82]	Open source	N/A	Byte code	Deadlocks, redundancy	Graphical	Not possible
6	SonarQube	[83]	Not Open Source	CWE	Source + bytecode	Duplicates, , bugs complexity errors etc.	Lists, charts	Possible
7	PMD	[56]	Open source	Sun Standards	Source code	Potential problems, Dead code, duplicate code and overcomplicated expressions,	List	Possible
8	Facebook Infer	[84]	Open Source	N/A	Source code	Null pointer, resource leaks, exceptions, annotation reachability,	List	Possible

2.3. Research Gap

This section deals with the research gap and gap in the proposed solution in industry automation domain. Code Analysis has been an active field of research for many years now. A lot tools have been proposed which can be seen in research and the market. After a detailed analysis of literature on Code Analysis we have identified 14 research papers in which Code analysis was done for Mission Critical Systems. Table 21 in Section 2.1.3 shows a summary of code analysis tools for different target systems, i.e. Mission Critical, Traditional Systems and General Systems. We further analyzed the tools targeted for Mission Critical Systems in Java only and a summary of our analysis is presented in Table 22. The frameworks proposed in these studies performed detection of errors including indexing, spell check, API misuse detection, Missing code, Assert, naming conventions, Fault Localization etc. However, it is evident from the analysis results that none of these frameworks is intended for the analysis of logical errors which can be fetal for mission critical systems. Furthermore, detection of concurrency related errors were also missed by these frameworks. These studies were further analyzed based on the standards that the frameworks are checking code compliance to. This help us identify another important gap in the literature i.e. none of the studies done so far has proposed a technique or framework for the analysis of code compliance to NASA JPL Java coding standards which are industry wide accepted standards, designed specifically for ground MCS.

Analysis from industrial perspective has been presented in Section 2.2 and a summary of industrial tools for code analysis of Java Language is presented in table 23. As per analysis results none of the tools checked errors related to logical errors and concurrency errors. These tools focused more on improving the maintainability of software rather than reliability of the software. Similarly, none of the industrial tools check code compliance to NASA's JPL standards. Only one tool Code Sonar was found to be detecting errors related to concurrency but it too does not checked code compliance to NASA's coding standards. Code Sonar implements a small set of JPL rules only for C language but not Java language.

The gap identified can be summarized in the following three major points:

1. Existing tools in the Industry focus on conformance to coding style and maintainability of System as compared to reliability of the software.

2. No tool available in literature or Industry that checks NASA’s JPL rules which are very important standards for Mission Critical Systems

3. No open source tool support in Literature and Industry for rules related to concurrency and logical errors in the code which are of major essence to real time systems

Hence, there is a need for open source tool that checks code compliance to NASA Standards related to Concurrency and Logical Errors, to improve software reliability of Mission Critical Real Time Systems.

Chapter 3

Proposed Methodology

CHAPTER 3: PROPOSED METHODOLOGY

As discussed earlier code analysis can greatly mitigate errors in the code at early stages of SDLC avoiding high maintenance cost at later stages. These errors can also sometimes prove to be fetal from safety or financial aspect. As MCS are mostly real time in nature therefor the main cause of its failure is concurrency and logical errors. NASA has provided coding standards for mission critical systems implemented in Java language but the implementation and checking of these standards has not been automated. A tool that checks code compliance to NASA's coding standards can greatly mitigate run time errors in the code and improve overall software reliability. Therefore we have proposed an open source tool that will detect concurrency and logical errors in the Java code by checking code compliance to NASA's Java coding standards.

The purpose of this chapter is to give a detailed description of the concepts used in the proposed solution. Sub section 3.1 presents our solution idea for the problem discussed in previous section. Sub-section 3.2 discusses our proposed system Workflow. Then NASA's selected rules and an example rules are presented in Sub-section 3.3 and 3.4 respectively. The Hybrid Code Analysis technique that our framework is based on is discussed in detail in Sub-section 3.5.

3.1. Solution Idea

Our Solution idea is to create a Framework for Code Analysis of Real Time Mission Critical Systems that:

- Ensure Software Reliability of RTMCS
- Automate JPL Standards by NASA
- Implement Concurrency and Logical errors related rules
- Extend language grammar with rules implementation
- Detect and locate potential causes of failure in code
- Suggest Solutions based on JPL Standards

3.2. Proposed System Workflow

A workflow diagram of the proposed system is presented in Figure 6. The workflow is explained in the following major steps.

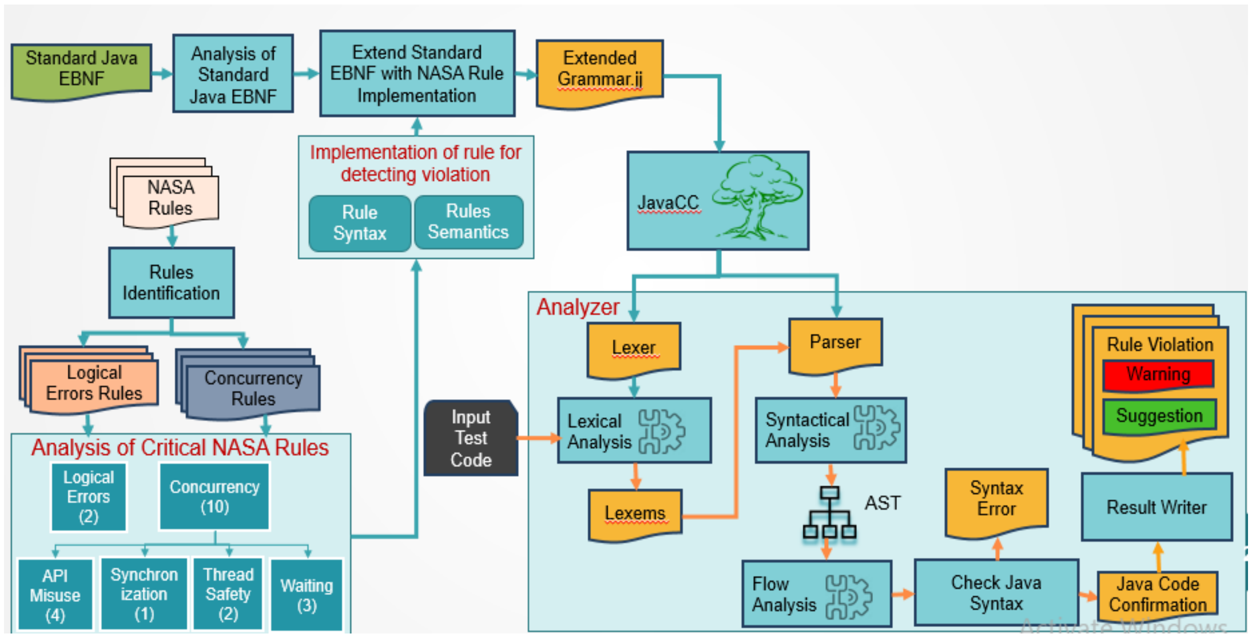


Figure 3.1: Proposed System Workflow

1. **Analysis of NASA Rules:**

Firstly, NASA's JPL Standards are analyzed to select rules that are to be implemented. First Critical Rules Category is selected. Then among the critical rules, the standards related to concurrency and logical errors are selected because these are the major cause of run time errors in real time systems. Concurrency rules are further divided into API Misuse, Synchronization, Thread Safety and Safety. We implemented four, one, two and three rules respectively from each subcategory. This makes a total of ten rules in concurrency category and two rules from logical error category. These rules are then implemented in Java Language based on the rule syntax and semantics.

2. Analysis of Standard Java EBNF:

In this step the standard Java EBNF in JavaCC format is analyzed. EBNF in JavaCC format has the lexical and semantic information together in one file with .jj extension.

3. Extension of Standard Java EBNF with implementation of NASA Rule:

In this step the standard Java EBNF resulting from step 2 is extended with the implementation of rules resulting from step 1. The output is an Extended Grammar .jj file.

4. Parsing using the JavaCC:

In this step the JavaCC parser takes in the extended grammar and creates a parser and lexer to parse input test code.

5. Analysis of the test Code

The input test code is first given as input to the Lexical Analyzer, generated by JavaCC. The lexer performs lexical analysis and breaks down the code into tokens or lexemes. The lexemes are then fed into the parser which performs syntactic analysis based on the extended grammar on the generated Extended Abstract Syntax Tree (AST). The extended tree is then given as input for Flow Analysis and all possible paths through the code are checked by creating Control Flow Graph (CFG). If according to analysis the code has any syntax error then the syntax error is displayed otherwise if the test code has rule violation the result writer displays the name of warning, location of warning and suggestion to fix the cause of possible error.

3.3. NASA's JPL Selected Rules

NASA's JPL have presented a set of coding standards for Ground Mission Critical Software implemented in Java language [63]. Major purpose providing the coding standards is to help programmer developing software in java language to mitigate run time errors in the code. This can highly increase software reliability of systems. Another, less focused purpose of these standards is to improve overall software quality factors such as maintainability and readability of code. These standards has been designed as a joint collaboration between the Semmle Limited and the

Laboratory of Reliable Software (LaRS) at the JPL, NASA. The rule set of Java Coding Standards has been divided into three main categories.

1) **Critical:** These rules are critical in nature i.e. following these rules is a must and violation of these standards must be fixed according to given suggestion at the highest priority.

2) **Important**: These rules are less critical in nature as compared to the critical rules and more critical than the advisory ones. Hence, these rules should be followed and any violation should be amended.

3) **Advisory:** Rules in this category include software conventions and good practices. Non-compliance to these rules are allowed but it is recommended to be avoided.

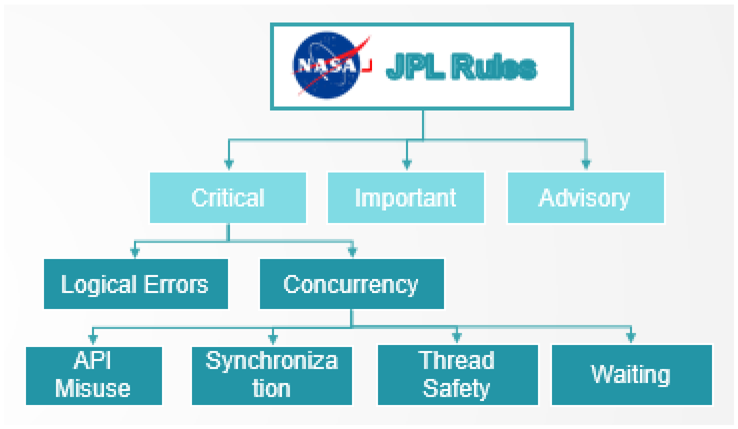


Figure 3.2: Classification of selected NASA's Rules

Since Mission Critical Software are mostly Real Time in nature, therefore, concurrency deadlocks and logical errors are the most common causes of failure in such system. Due to this reason we have selected rules related to concurrency and logical errors for automation in our framework. Secondly, these rules improved the reliability of software, rather than maintainability of software, which is the aim of this research. Furthermore, we have selected rules only from the critical category for a start, since those are more important. Selected rules are given below:

1. Critical Rules
    1.1. Concurrency
        1.1.1. API Misuse
            1.1.1.1. Avoid setting thread priorities
            1.1.1.2. Avoid using 'notify'

1.1.1.3. Do not call 'Thread.yield'

1.1.1.4. Do not start a thread in a constructor

1.1.2. Synchronization

1.1.2.1. Avoid empty synchronized blocks

1.1.3. Thread Safety

1.1.3.1. Avoid static fields of type 'DateFormat' (or its descendants)

1.1.3.2. Ensure that a method releases locks on exit

1.1.4. Waiting

1.1.4.1. Avoid calling 'Object.wait' while two locks are held

1.1.4.2. Avoid calling 'Thread.sleep' with a lock held

1.1.4.3. Avoid calling 'wait' on a 'Condition' interface

1.2. Logic Errors

1.2.1. Avoid array downcasts

1.2.2. Do not call a non-final method from a constructor

3.4. Example Rule

Rule:

Ensure that a method releases locks on exit.

Rule Category:

Critical→ Concurrency → Thread Safety

Description of Rule:

Methods that acquire a lock and do not release the lock in some of the exiting path from the method can result in a deadlock

Recommendation/ Suggestion

Ensure that all exit paths of the method release the lock.

Example

In the given example a lock is acquired at line 5 in run method. The lock is released inside the if condition. However of the condition is not met then the lock is never released. This can cause a deadlock if any other method acquires the same lock.

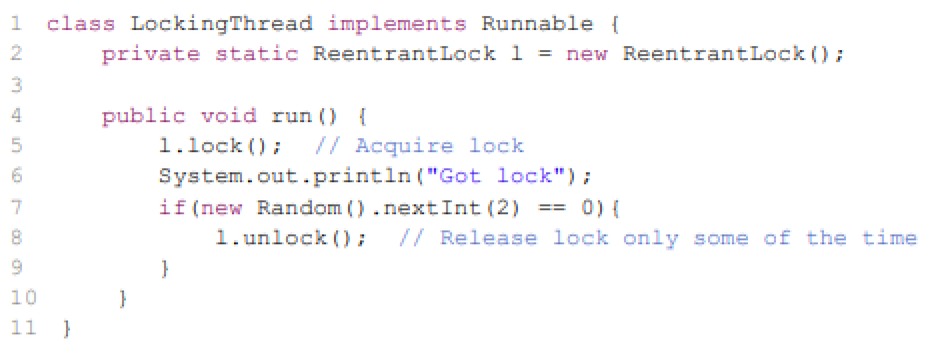

```
class LockingThread implements Runnable {
    private static ReentrantLock l = new ReentrantLock();

    public void run() {
        l.lock();  // Acquire lock
        System.out.println("Got lock");
        if(new Random().nextInt(2) == 0){
            l.unlock();  // Release lock only some of the time
        }
     }
 }
```

Figure 3.3: Example Code

To avoid this problems there should be a thread unlock in each of the exiting paths,

3.5. Hybrid Analysis Technique

Code Analysis ensures early corrections of code by finding potential trouble spots such:

- Logical Errors
- Unused code
- Code clones
- Concurrency errors
- Security Vulnerabilities
- Deadlocks etc.

It is normally done early on in the software development life cycle which helps detect and correct flaws in the software which can become very expensive to maintain in the later stages. The techniques employed for static code analysis range from elementary approaches like pattern based approaches to more complex ones like Syntactic Analysis, Flow Analysis, Taint Analysis and Machine Learning. One or more techniques can be combined to form a **hybrid** which has proven to be more effective. As discussed before in Chapter 1, Syntactic Analysis and Flow Analysis are two most common methods used for Code Analysis. Besides these techniques also give the best detection results. Therefore, we have used a combination of the two techniques, i.e. Syntactic and Flow Analysis and formed a hybrid techniques for better detection results.

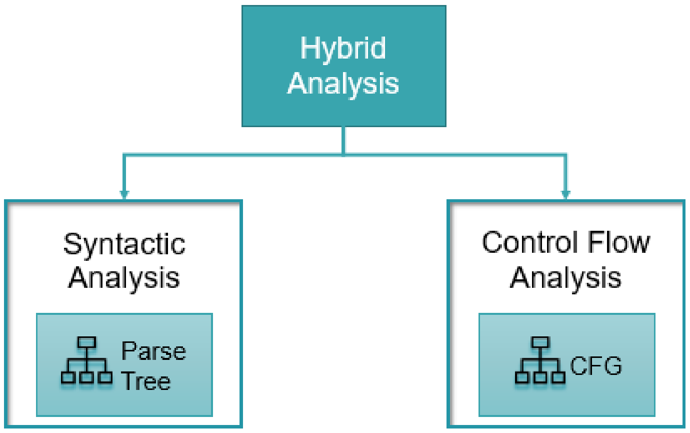


Figure 3.4: Hybrid Analysis

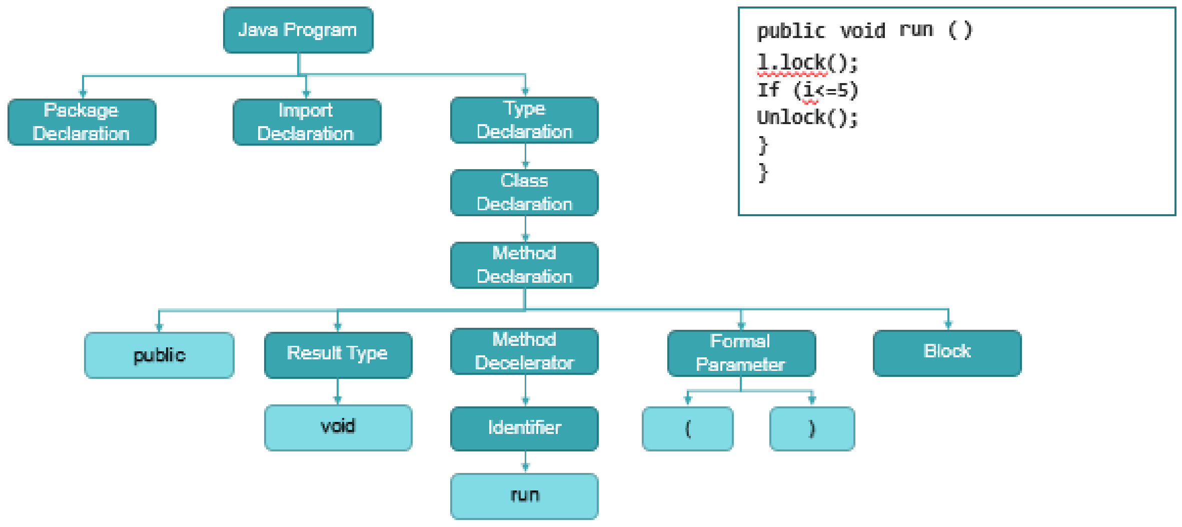
3.5.1.Syntactic Analysis

Syntactic Analysis is based on imposing rules, implied by a context-free grammar, on syntax tree or program. It can be:

- **Top down:** In this approach the parse tree starts creating from the top i.e. the root and proceeds towards the bottom i.e. the leaves.
- **Bottom- down:** In this approach the parse tree is constructed from the leaf and proceeds towards the roots.

Using top-down parsing has advantages over Bottom down approach such as use of more general grammar, easier to debug, and passing values (attributes) both up and down the parse tree. To make use of Top down parsing we have used the JavaCC parser, which parses code according to grammar in a top down manner. Our framework parses our extended version of Java Grammar to detect anomalies in the code. A parse tree is created as a result of parsing the code. The parse tree is a rooted tree like structure that represents the syntactic structure of the input code. Considering the example given in Section 3.4. The input code is first converted to lexemes by JavaCC lexical Analyzer. These lexemes are then converted into a parse tree based on grammar file. Figure 10 presents the parse tree of first line of the example code i.e.

Line no 1: public void run();



```
public void run ()
l.lock();
If (i<=5)
Unlock();
}
}
```

Figure 3.5: Parse Tree of Example Code

The non-terminal nodes are the non-leaf nodes of the parse tree whereas terminals are presented by leaf nodes. Terminals are the token of the input code. The method body comes under 'Block' node of the Method Declaration node's children. The method 'Block' node and its children are presented in Figure 11. This figure represents parse generated for second and third line of code i.e.

Line no 2:

{ l.lock();

Line no 3:

if (i<=5)

The body of 'if' statement comes under the 'Block' node which is the child node of 'Statement' node which in turn is the child node of 'IfStatement' node. This block and its children is presented in Figure 12. The figure shows parse tree for 'if' body i.e:

Line no 4:

{ unlock();}

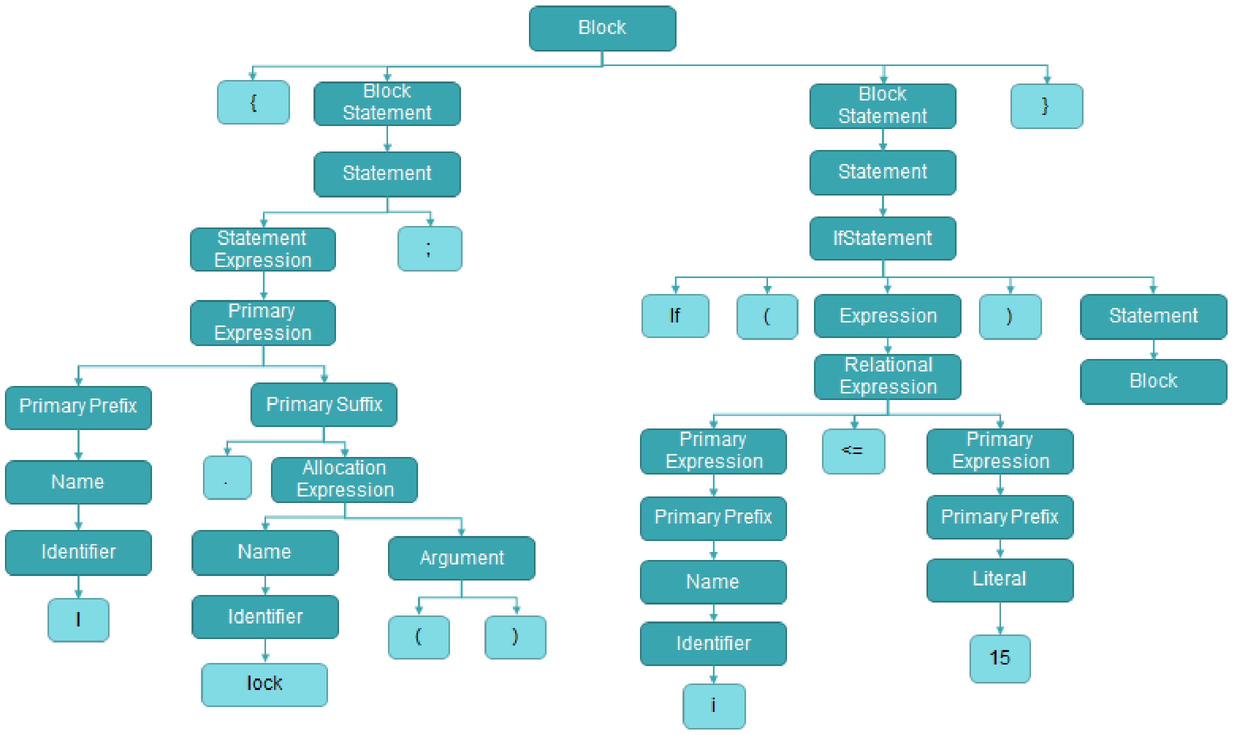


Figure 3.6: Parse Tree of Example Code Method Block

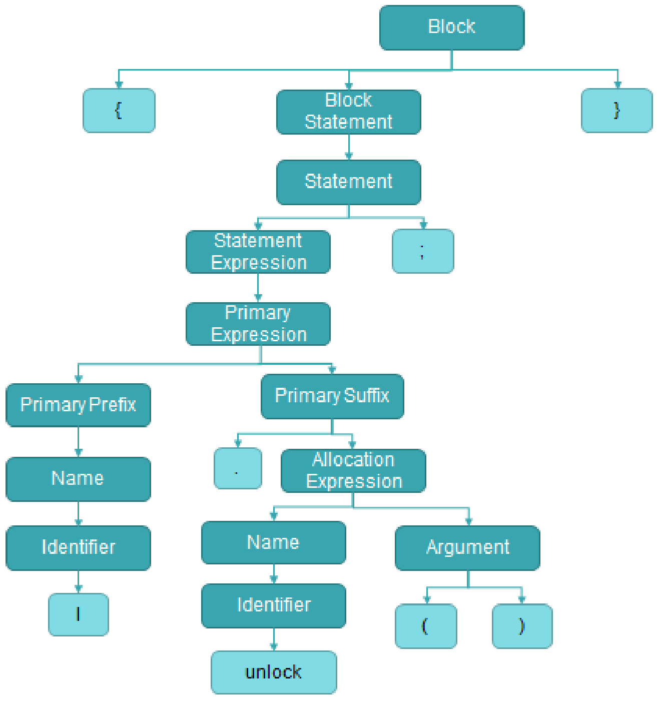


Figure 3.7: Parse Tree of Example Code if Block

3.5.2.Flow Analysis

Flow analysis ensures analysis of control or data flow around the system using graphs such as Control Flow Graph (CFG), Data Flow Graph (DFG) and Call Graph. Our framework uses Flow Analysis technique by checking CFG of the input code in combination with Syntactic analysis for embedding flaw checks in Grammar of the code.

Flow Analysis of Example Rule

Flow analysis is important so that all possible paths through the code are checked during code analysis. If we consider the example given in sub section 3.4, the problem occurs mainly because flow through the program i.e. if 'if; statement is true' is correct while other flow through the program may give a logical error. Figure 13 shows three possible flows of a program having i.e.

Simple Flow: In this flow there is no alternate path. The program starts, thread is locked, thread is unlocked and finally program exits.

If Else Conditional Flow: In this scenario program starts then there is a thread lock, then there are two possible paths through a program based on Condition node and a thread unlock is present in both conditions. Finally the program exits.

Case Condition: In this scenario the program starts, then there is a thread lock, followed by a Case Condition. Based on this condition there can be two or more paths exiting from the case condition. A thread unlock is present in each of the case condition. Finally the program exits.

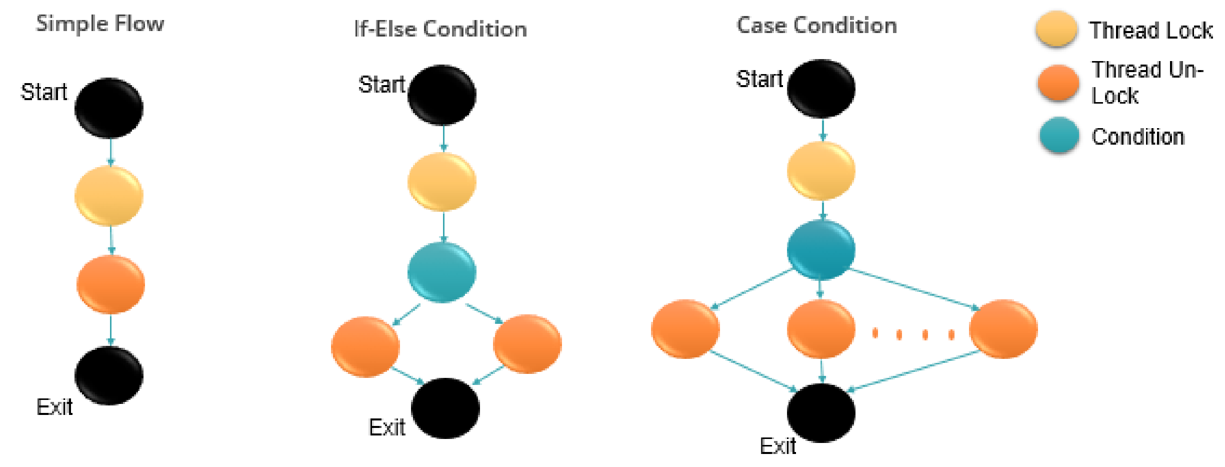


Figure 3.8: CFG of Simple Flow, If-Else Conditional Flow and Case Condition Flow

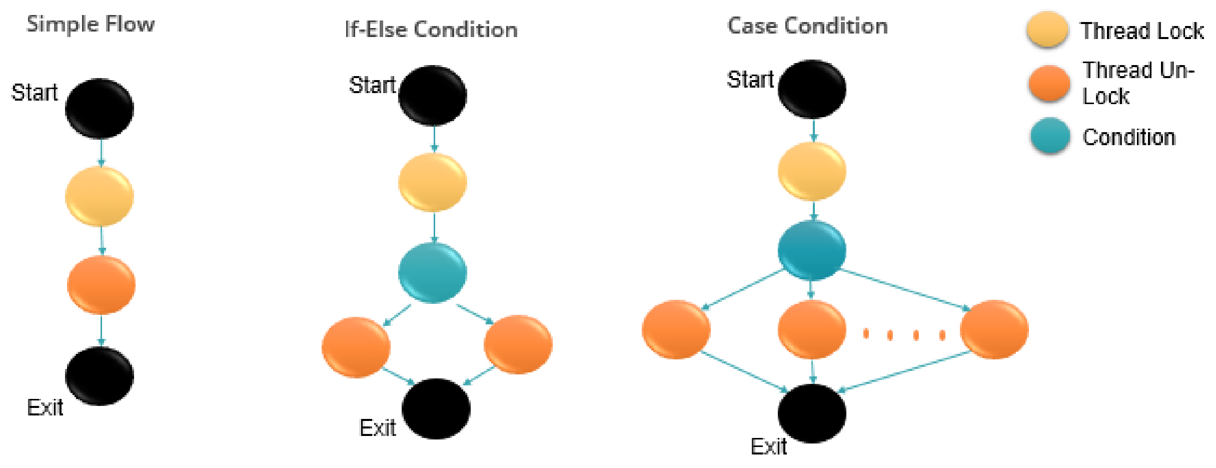


Figure 3.9: Extended CFG of Simple Flow, If-Else Conditional Flow and Case Condition Flow

Extended Control Flow Graph

To detect warning in the given three scenarios mentioned above, there is a need to check if the thread is unlocked before exit point of the program. Therefore we have extended our CFGs which a check point (shown in green) right before the exit point (Figure 14). The purpose of this check point is to check if the thread that was locked is unlocked or not.

Flow Analysis of Extended CFG

With the extended CFGs the flow of program in three different scenarios is given below and also presented diagrammatically in Figure 15.

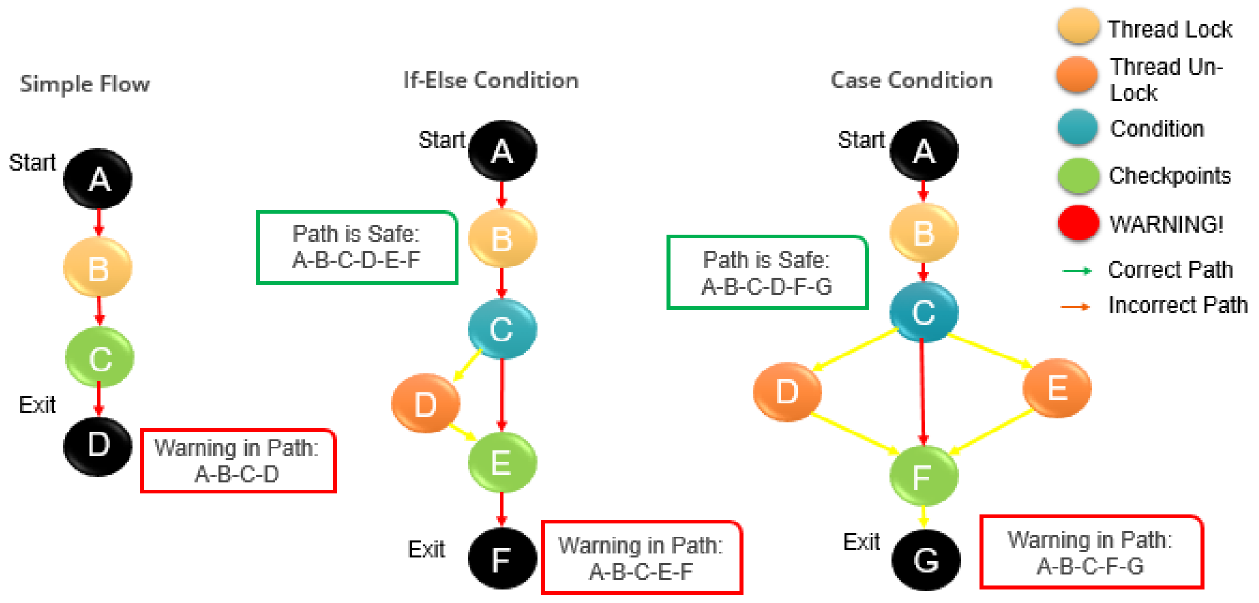


Figure 3.10: Warning path in CFG of Simple Flow, If-Else Conditional Flow and Case Conditional Flow

Simple Flow: In this flow there is only one path i.e. A-B-C-D. If there is a thread unlock after a thread lock then the path will be safe. However, if the thread unlock node is missing, as in figure 16, the check point node ‘C’ will call a warning.

If Else Conditional Flow: In this scenario suppose there are two paths in the program based on whether the Condition (node ‘C’) is true or false.

- **Path 1 (A-B-C-D-E-F):** This path is safe because the thread is unlocked inside the condition at node ‘D’ after being locked at node ‘B’.
- **Path 1 (A-B-C-E-F):** In this path there is a thread lock at Node ‘B’ but no thread unlock before the checkpoint at Node ‘E’. Therefore checkpoint will call a warning.

Case Condition: In this scenario the program starts, then there is a thread lock, followed by a Case Condition. Based on this condition there can be two or more paths exiting from the case condition. Based on Condition there are three paths.

- **Path 1 (A-B-C-D-F and 2 (A-B-C-E-F):** This path is safe because the thread is unlocked inside the condition at node ‘D’ after being locked at node ‘B’.
- **Path 3 (A-B-C-F):** In this path there is a thread lock at Node ‘B’ but no thread unlock before the checkpoint at Node ‘E’. Therefore checkpoint will call a warning.

Based on flow analysis the checkpoint in extended EBNF based parsed tree checks if a locked thread is unlocked before Exit point or not. While traversing the parse tree a lock flag is raised when the thread is locked. Similarly a condition or thread unlock flag is raised when it reaches respective nodes in the parse tree. If the thread unlock flag is not raised before the exit point it will prevent any other thread from starting. Therefore, the checkpoint will call a warning. This example is shown diagrammatically in Figure 16.

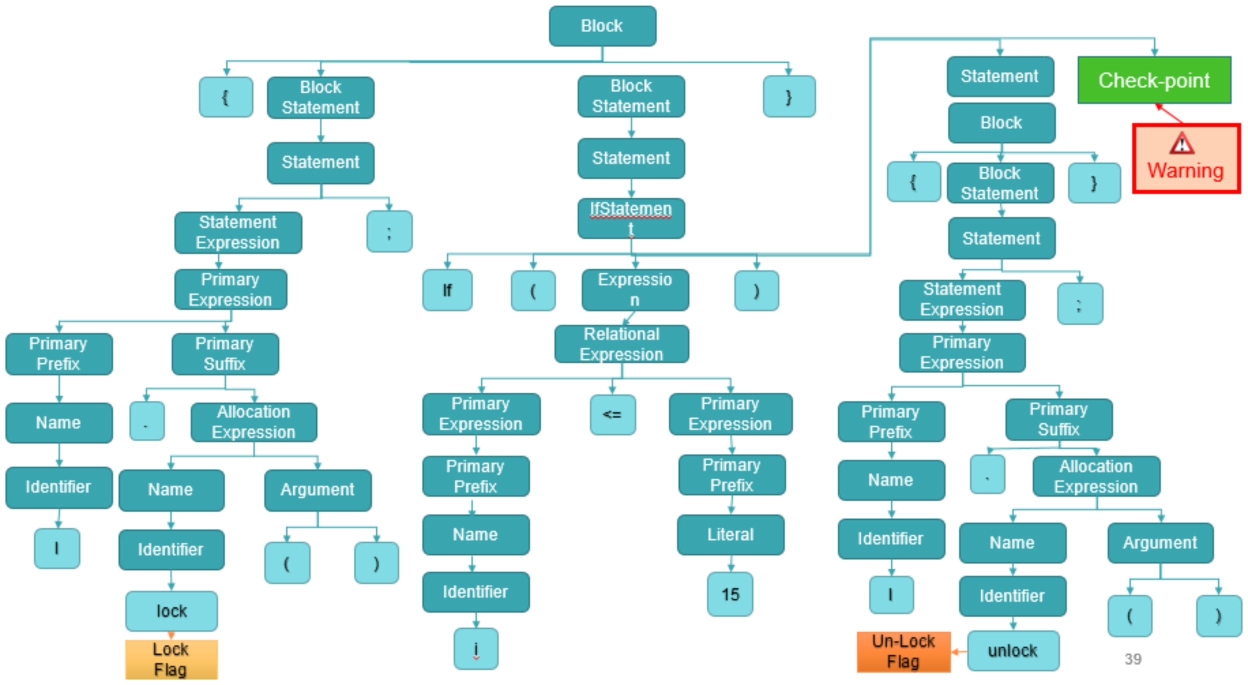


Figure 3.11: Extended Parse Tree

Chapter 4

Implementation

CHAPTER 4: IMPLEMENTATION

This chapter provides the implementation details of our proposed framework. Section 4.1 describes the Architecture of our framework. Section 4.2 presents a description of the parser we have used i.e. JavaCC. How Java Grammar is extended to embed NASA's rules is described in Section 4.3. Finally, the tool interface along with description is presented in Section 4.4. Our tool is open source and can be found here [78]

4.1. MCS Code Analyzer Architecture

Tool support is an important factor to increase the productivity of software development. A tool support architecture to support the framework is shown Figure 17. The architecture comprise of two main components i.e. Eclipse Integrated Development Environment (IDE) with its Plugins and the Analyzer having different analysis Applications. Latest Version of Eclipse i.e.2020-6 is used for the development of the framework. Eclipse IDE is usually used to develop application in Java Language but it can also be used to develop application in other languages such as JavaScript, C#, PHP etc. JavaCC, the primary tool used in our framework, is installed as a plugin in Eclipse IDE. Windows Builder plugin is installed to build the front end of the application. XML (Wide Web Dev) plugin is used by the framework for generating XML results. The Analyzer component comprises of three basic techniques i.e. the Lexical Analysis, Syntax checking and Hybrid Analysis. Lexical Analyzer and the Syntax checker are generated by the JavaCC based on a set of Grammar rules. Hybrid Analysis is further comprised of two techniques .i.e. Syntactic Analysis and Flow Analysis. These two techniques are discussed in detail in Sub-Section 3.5.

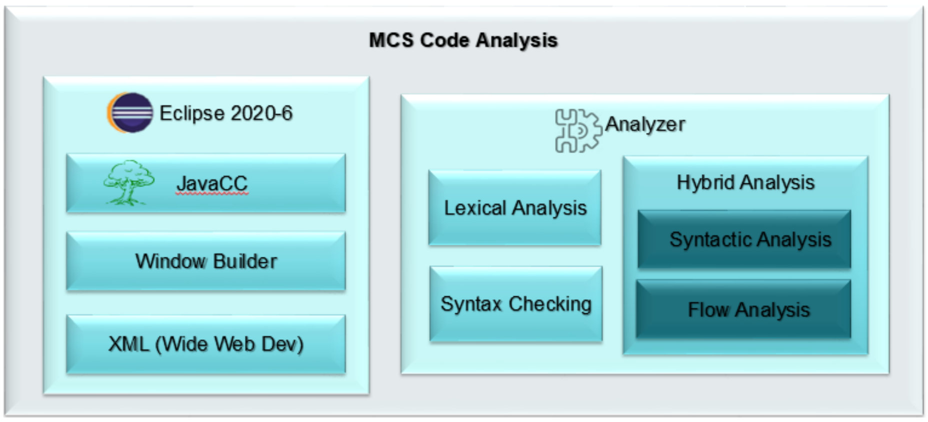


Figure 4.1: Architecture Diagram for Tools and Techniques

4.2. JavaCC

Java Compiler Compiler (JavaCC) is the most popular parser generator. Its design is shown in Figure 18. Firstly, a sequence of character is given as input into the Token Manager. The Token Manager creates tokens based on some grammar rules. The next component is the parser which takes the generated tokens as input, analyses its structure, and creates a parse tree or other user defined structure based on some grammar.

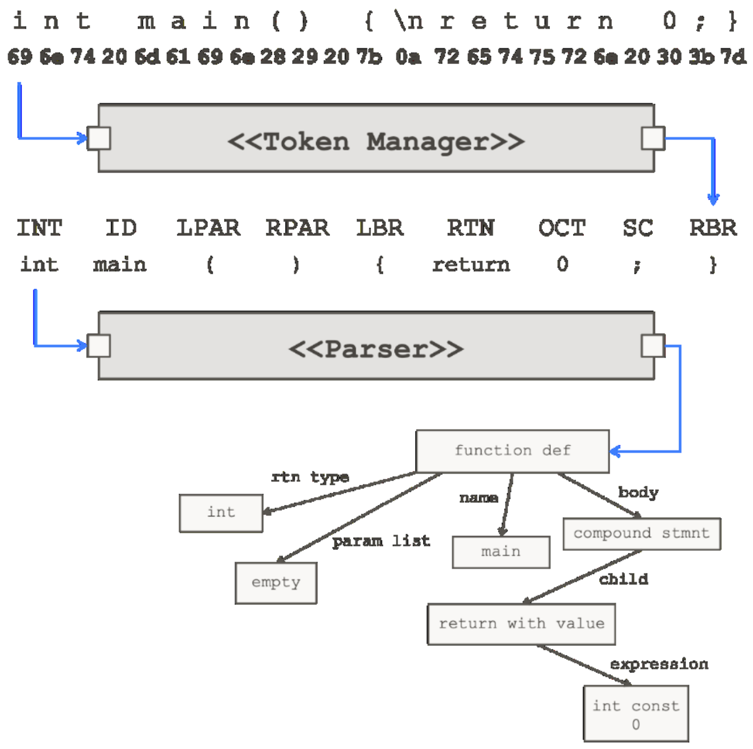

i n t   m a i n ( )   { \n r e t u r n   0 ; }
69 6e 74 20 6d 61 69 6e 28 29 20 7b 0a 72 65 74 75 72 6e 20 30 3b 7d
<<Token Manager>>
INT ID LPAR RPAR LBR RTN OCT SC RBR
int main ( ) { return 0 ; }
<<Parser>>
function def
rtn type
int
name
body
param list
main
compound stmnt
empty
child
return with value
expression
int const 0

Figure 4.2: JavaCC Design

Some major features of JavaCC are:

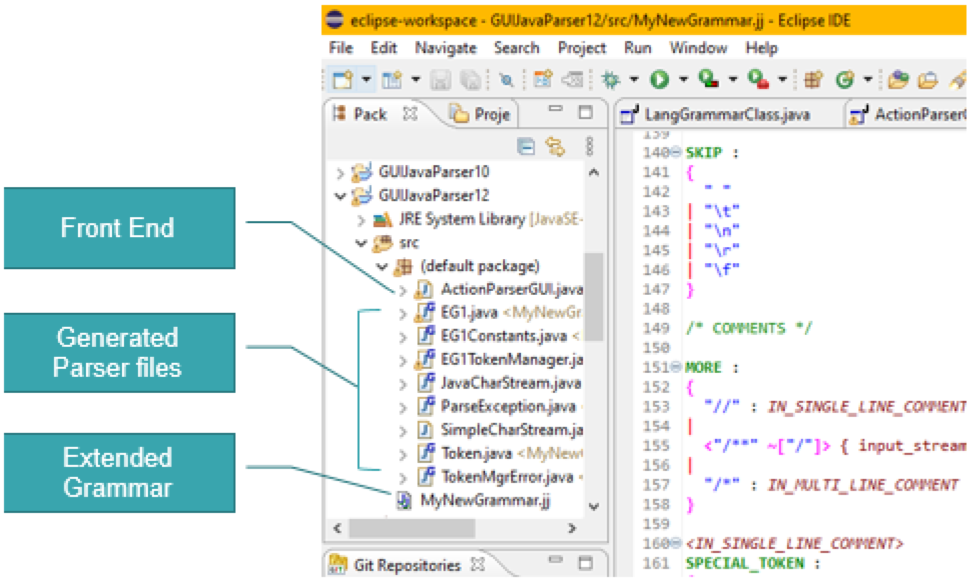
Top-Down approach: The parser created by JavaCC uses top-down (Recursive descent) as compared to YACC which uses bottom up parsing. The benefit of top down parsing is that more general grammar can be used, it is easier to debug and it has the ability to parse down to any non-terminal in the grammar and pass values both up and down the parse tree.

Tool Support. Being the most popular parser, JavaCC has by far the largest user community and tool support.

Flexible: The tool is highly customizable.

Lexical Specifications: The tool has the BNF grammar rules and lexical rules such as regular expression in the same file.

JavaCC project has a .jj file. This file contains the Context Free Grammar. Based on this grammar the JavaCC generates the parser implemented in files including Token manager, Simple Character Stream, Token, Token Manager Error, Constants, Character Stream, Parser Exception and the parser file itself. Figure 19 shows these automatically generated files along with the the front end file and .jj grammar file which has our extended grammar.



```
SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}

/* COMMENTS */

MORE :
{
  "//" : IN_SINGLE_LINE_COMMENT
|
  <"/**" ~["/"]> { input_stream
|
  "/*" : IN_MULTI_LINE_COMMENT
}

<IN_SINGLE_LINE_COMMENT>
SPECIAL_TOKEN :
```

Figure 4.3: JavaCC files

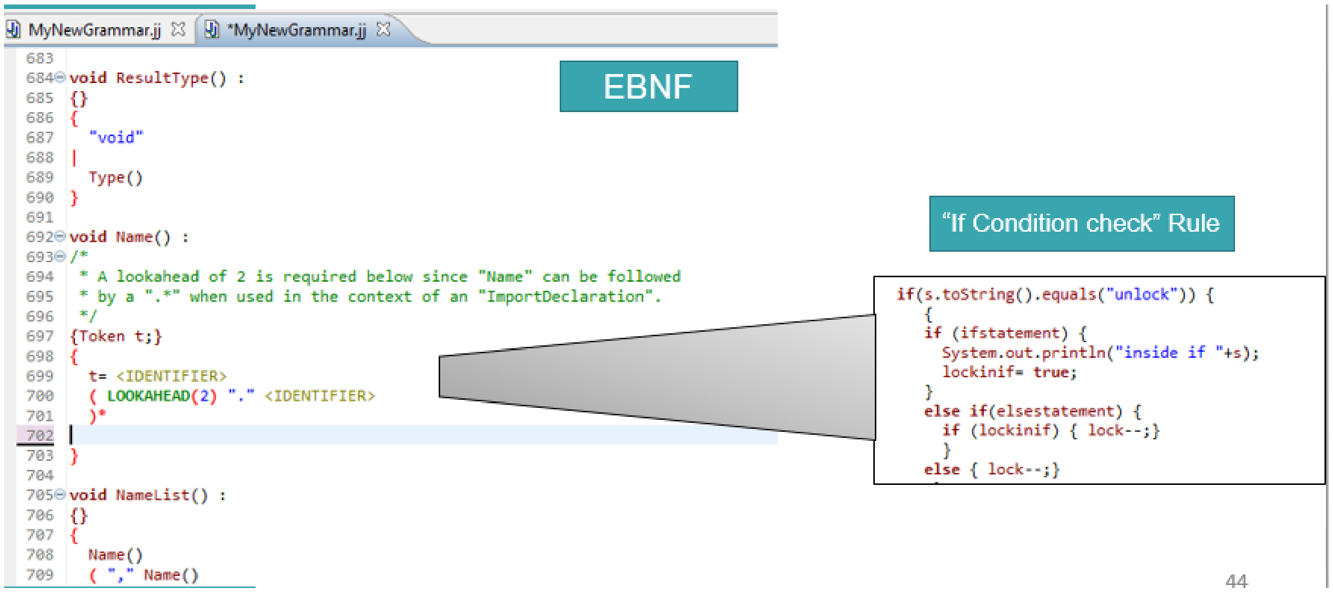
4.3. Embedding NASA's Rules in EBNF Rules

We have extended the rules of Java Language grammar with rules of NASA. JavaCC parser creates implements the grammar and creates parser files that analyzes the test code. For ease in understanding, consider example rule discussed before in sub section 3.4 again. The example code shows that a thread that is locked is unlocked in if statement but not in the else statement due to which the next thread cannot be started. Therefore in the grammar we have added flags where a lock, unlock, if statement, else statement, case statement is detected. Within the limits of 'if statement' the 'ifFlag' is raised similarly flags for else statement and case statement is raised inside the scope of else statement and case statements respectively. The algorithm for simple if condition check is given below:

Algorithm:

```
If unlockFlag then
   If ifFlag then UnlockInIf=true
If elsefalg is raised then
   If unlockinif is true then unlocked=false
   Else unlocked = true
If unlocked is false then generate warning
```

The flags and checks are embedded in the grammar .jj file shown in Figure 20.



```
void ResultType() :
{}
{
  "void"
|
  Type()
}

void Name() :
/*
 * A lookahead of 2 is required below since "Name" can be followed
 * by a ".*" when used in the context of an "ImportDeclaration".
 */
{Token t;}
{
  t= <IDENTIFIER>
  ( LOOKAHEAD(2) "." <IDENTIFIER>
  )*

}

void NameList() :
{}
{
  Name()
  ( "," Name()
```

```
if(s.toString().equals("unlock")) {
   {
   if (ifstatement) {
     System.out.println("inside if "+s);
     lockinif= true;
   }
   else if(elsestatement) {
     if (lockinif) { lock--;}
     }
   else { lock--;}
```

Figure 4.4: Embedding NASA's rules in Java Rules

4.4. Tool Interface

The main Tool Interface is given in Figure 21. The input Test code can be entered in the upper text area under the 'Enter Code' and the analysis results are shown in the lower text area under the 'warning'. The '**Check**' button is used start analysis of the text given in input text area. The '**Reset**' button clears everything in the input text area and the result area. The **'Save Result'** button saves the result displayed in warning text area, in user system in xml format. The result file saved in xml format is shown in figure 22.

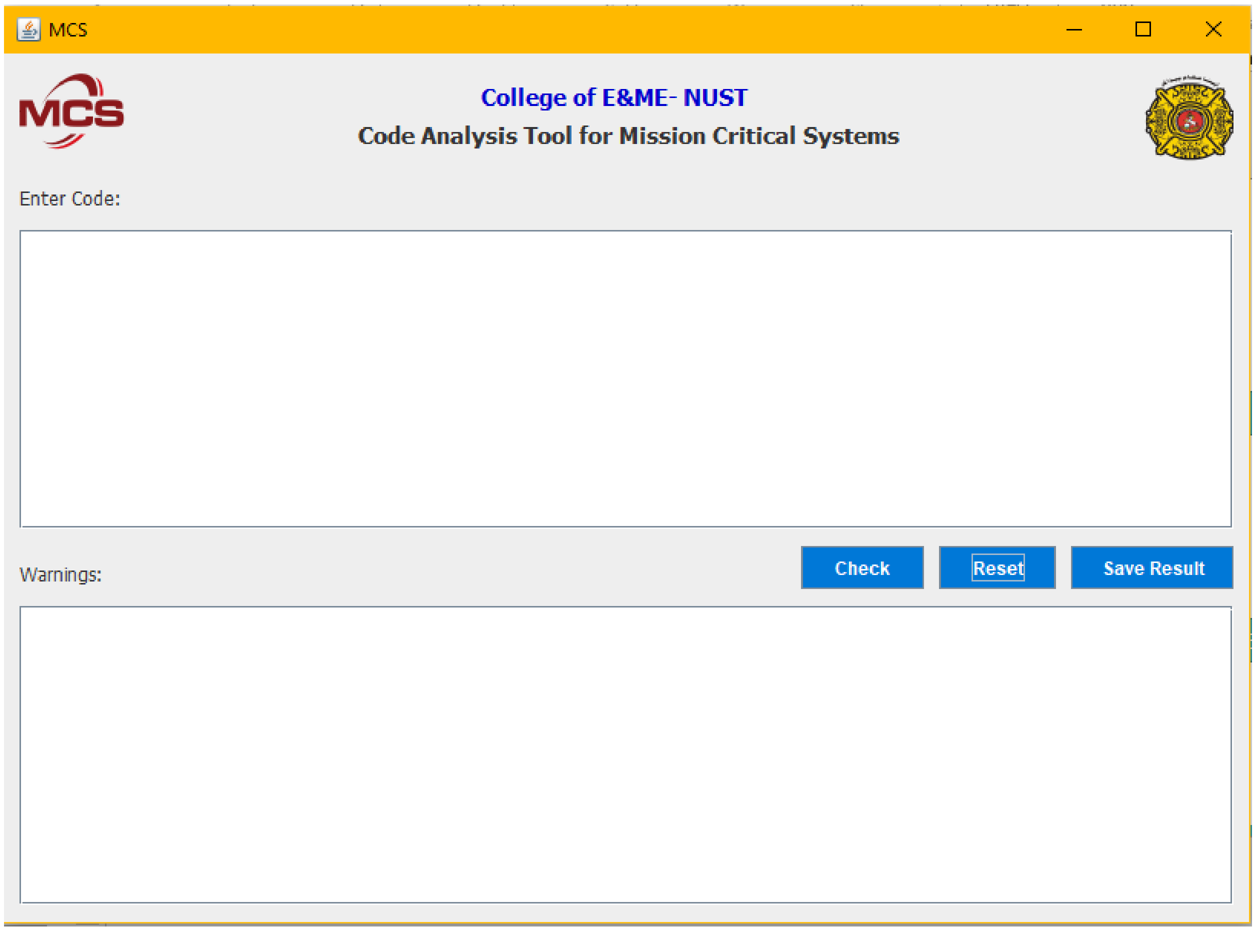
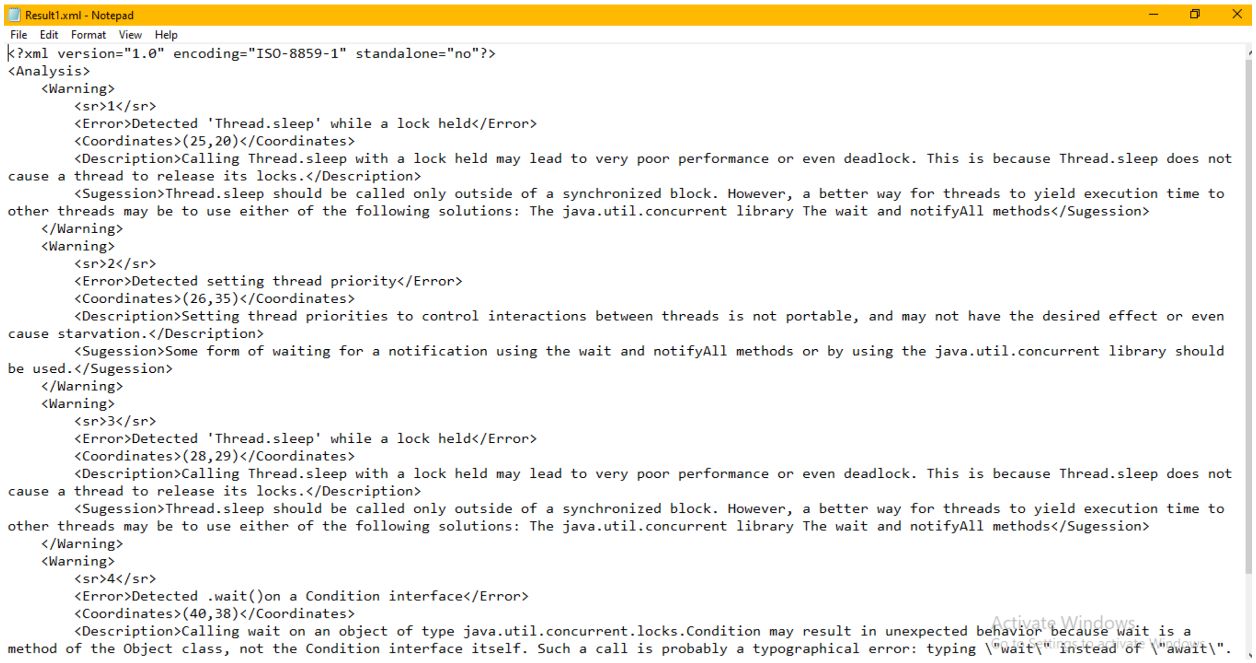


Figure 4.5: Tool Main Interface

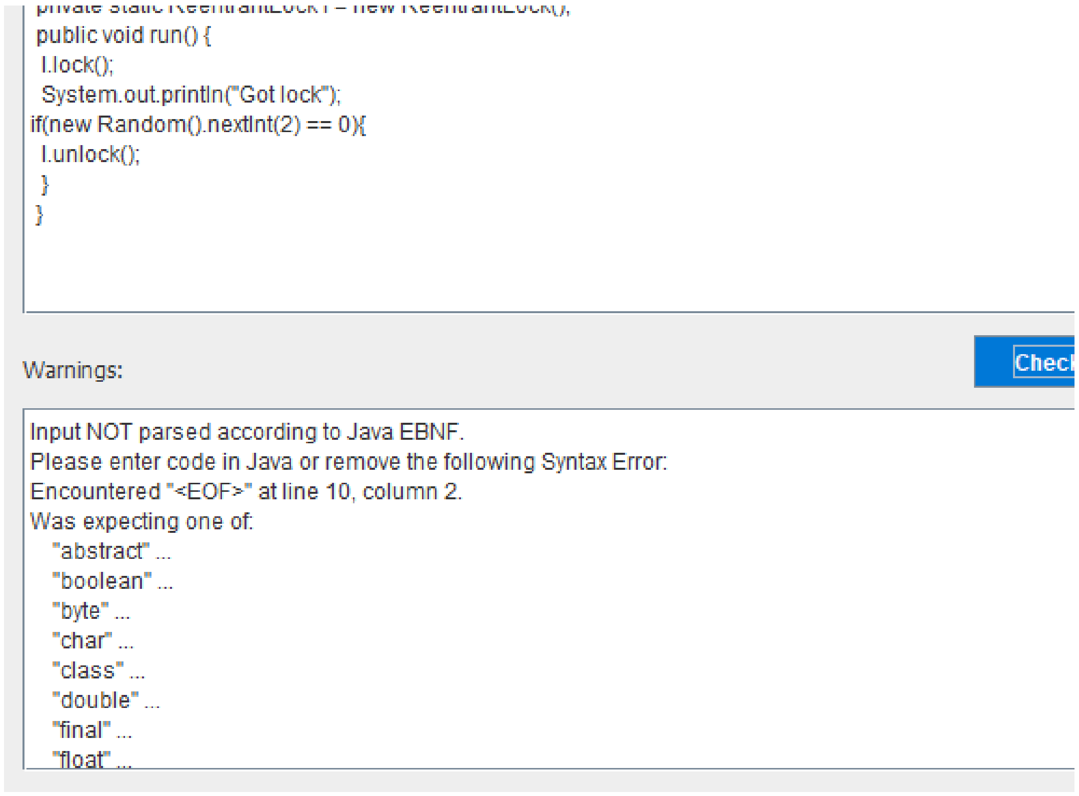

Result1.xml - Notepad
File Edit Format View Help
Activate Windows
Go to Settings to activate Windows.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<Analysis>
    <Warning>
        <sr>1</sr>
        <Error>Detected 'Thread.sleep' while a lock held</Error>
        <Coordinates>(25,20)</Coordinates>
        <Description>Calling Thread.sleep with a lock held may lead to very poor performance or even deadlock. This is because Thread.sleep does not
cause a thread to release its locks.</Description>
        <Sugession>Thread.sleep should be called only outside of a synchronized block. However, a better way for threads to yield execution time to
other threads may be to use either of the following solutions: The java.util.concurrent library The wait and notifyAll methods</Sugession>
    </Warning>
    <Warning>
        <sr>2</sr>
        <Error>Detected setting thread priority</Error>
        <Coordinates>(26,35)</Coordinates>
        <Description>Setting thread priorities to control interactions between threads is not portable, and may not have the desired effect or even
cause starvation.</Description>
        <Sugession>Some form of waiting for a notification using the wait and notifyAll methods or by using the java.util.concurrent library should
be used.</Sugession>
    </Warning>
    <Warning>
        <sr>3</sr>
        <Error>Detected 'Thread.sleep' while a lock held</Error>
        <Coordinates>(28,29)</Coordinates>
        <Description>Calling Thread.sleep with a lock held may lead to very poor performance or even deadlock. This is because Thread.sleep does not
cause a thread to release its locks.</Description>
        <Sugession>Thread.sleep should be called only outside of a synchronized block. However, a better way for threads to yield execution time to
other threads may be to use either of the following solutions: The java.util.concurrent library The wait and notifyAll methods</Sugession>
    </Warning>
    <Warning>
        <sr>4</sr>
        <Error>Detected .wait()on a Condition interface</Error>
        <Coordinates>(40,38)</Coordinates>
        <Description>Calling wait on an object of type java.util.concurrent.locks.Condition may result in unexpected behavior because wait is a
method of the Object class, not the Condition interface itself. Such a call is probably a typographical error: typing \"wait\" instead of \"await\".
```

Figure 4.6: Saved XML results

Syntax Checking Results

As discussed in the Methodology chapter, before checking input code compliance to rules, our framework first checks if the input code is in Java and has correct syntax. To check if the code is syntactically as per Java Language, the code is analyzed against Java Language EBNF. If input code is in Natural Language or any other language, then the message "Input not parsed according to Java EBNF. Please Enter Java Code or remove the following Syntax Error" is displayed. In Figure 19, there is a syntax error in the input Test code. A bracket '{' is missing in the code. Therefore, warning is generated that the input is not successfully parsed according to Java EBNF. Furthermore, it also tells where the syntax error lies and what is expected at the location. This can give user an idea as to what could possibly be wrong with the input code.



```
 public void run() {
  l.lock();
  System.out.println("Got lock");
if(new Random().nextInt(2) == 0){
  l.unlock();
  }
 }
```

```
Input NOT parsed according to Java EBNF.
Please enter code in Java or remove the following Syntax Error:
Encountered "<EOF>" at line 10, column 2.
Was expecting one of:
    "abstract" ...
    "boolean" ...
    "byte" ...
    "char" ...
    "class" ...
    "double" ...
    "final" ...
    "float" ...
```

Figure 4.7: Syntax Checking Result

Code Analysis Results

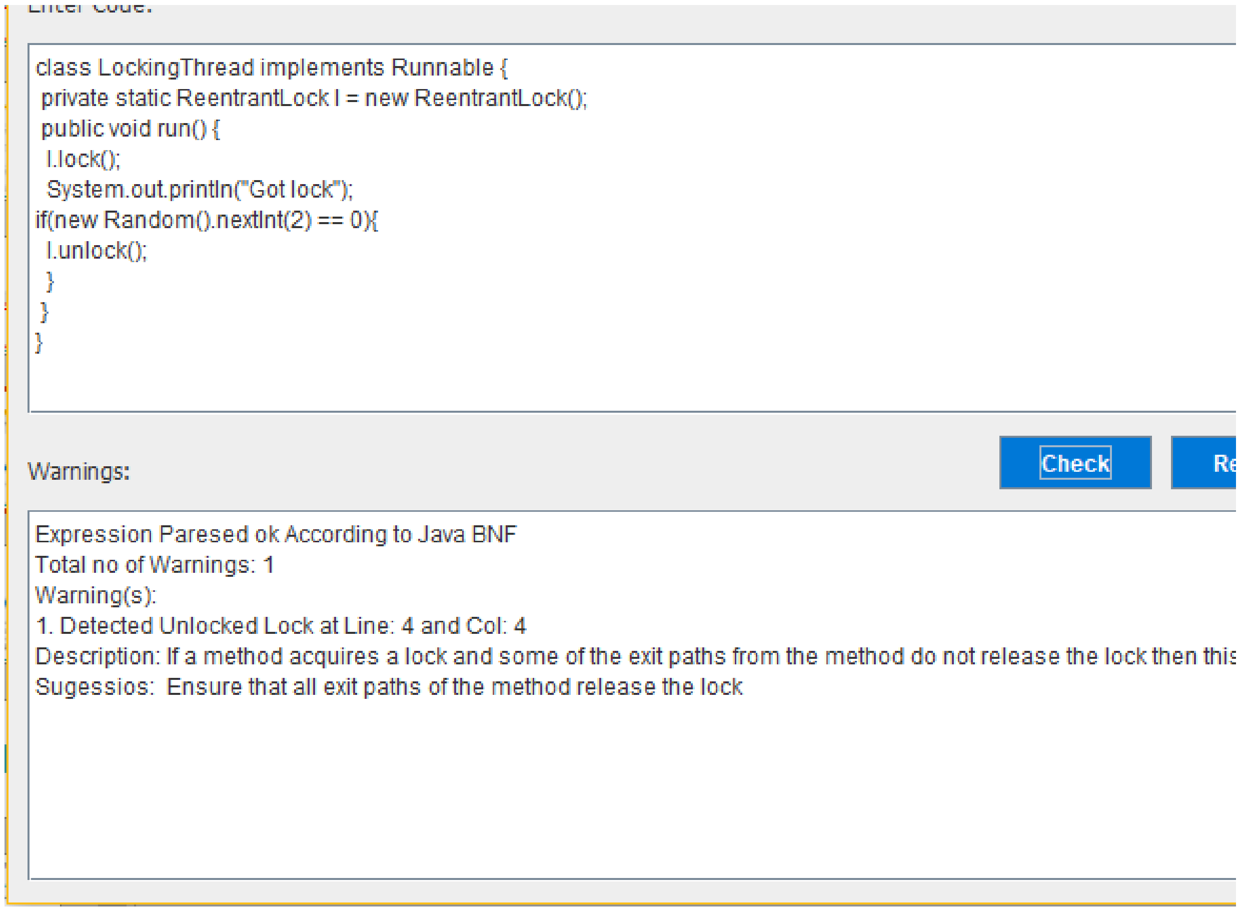
If the input code is in Java and has no syntax errors then the message "Expression Parsed ok According to Java BNF" is displayed. In this case following information is displayed.

1) Total No of Warnings: Is the total number of warning in the input code. It can be 0 in case no warning is detected.

2) List of warning(s): If there is in warning in the code this list is shown empty otherwise a list of warnings is shown here. The list contains the following information about each warning.

- **Warning Name:** It is the name of warning.
- **Warning Location:** It is the line number and column number of where the warning is located in the input code.
- **Warning Description:** It is a small description of the warning.
- **Suggestion(s):** It is a possible fix to avoid the warning as per NASA's Standards.

Figure 24 presents the result of analysis of input code discussed in Example Rule discussed in Sub-Section 3.4. The tool correctly identifies the unlocked lock at Line 4 and Column 4 of the input code.



```
class LockingThread implements Runnable {
 private static ReentrantLock l = new ReentrantLock();
 public void run() {
  l.lock();
  System.out.println("Got lock");
if(new Random().nextInt(2) == 0){
  l.unlock();
  }
 }
}
```

```
Expression Paresed ok According to Java BNF
Total no of Warnings: 1
Warning(s):
1. Detected Unlocked Lock at Line: 4 and Col: 4
Description: If a method acquires a lock and some of the exit paths from the method do not release the lock then this
Sugessios:  Ensure that all exit paths of the method release the lock
```

Figure 4.8: Example Rule Warning Results

Chapter 5

Validation

CHAPTER 5: VALIDATION

This section presents the validation of our proposed framework with the help of some open source projects. Section 5.1 discusses the validation procedure and results. Section 5.2 and its subsection discusses the in detail validation in three of the test projects in its subsections

5.1. Validation Process

The framework is validated by analyzing 12 mission critical open source projects. Firstly, five Standard Java MCS projects are analyzed using our framework. Standards projects details are presented in Table 25. Reference means the link to the source of the project. Files means the number of Java Files in the project. SLOC means the Java source line of code in the project.

Table 5.1: Standard Project Details

SR#	Standard Project Name	Reference	Files	SLOC
1	Tele Health Care System	[74]	24	2877
2	Autonomous Driving	[67]	5	355
3	Flight Navigation	[68]	11	622
4	Online Analytical Processing (OLAP) Server	[70]	32	2231
5	Automobile Cruise Control System	[69]	17	3860

To further validate our framework we have analyzed 7 projects after inducing errors in it. Details of projects with errors induced are given in Table 26.

Table 5.2: Details of Projects with Error Induced.

SR#	**Project**	**Link**	**Files**	**SLOC**
1	Flight Control System:	[74]	11	651
2	Bank Customer Multi-Threaded Project	[65]	4	268
3	Chat server	[66]	3	602
4	Parent monitor	[75]	17	1098
5	Elastic Cloud Computing using Multi-threading	[77]	4	430
6	Hadoop distributed processing Project	[78]	1	112
7	Elevator Control System	[64]	7	501

5.2. Test Projects Details

Mission Critical Systems in which multi-threading is used is selected to test the validity of our framework. The projects are downloaded from GitHub and Source forge.

1.5.1. Test Project 1: Elevator Control System

Description of Project:

The Elevator Control System [64] simulates an elevator system in a building with five floors. For each floor a separate elevator thread is executed. A BuildingManager has access to each elevator and keep information about elevator state and floor. Each elevator can detect if a passenger is waiting at any given time for every floor in the building through the BuildingManager. In order to avoid race conditions while accessing the BuildingManager object by the Elevator of different floors, Java Synchronized method is used in Elevator threads.

Description of Warning Induced:

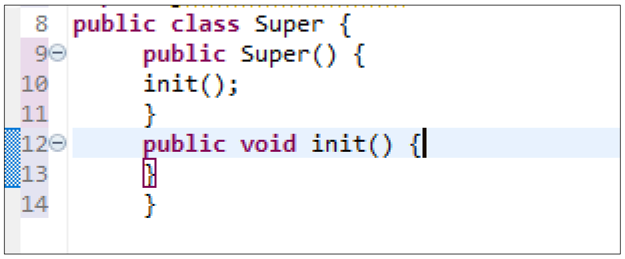
We have induced three different warnings in this project based on the semantics of the code. The violations are:

1) Call to a non-final method from a constructor

2) Creating static field of DateTime type

3) Creating Empty Synchronized block

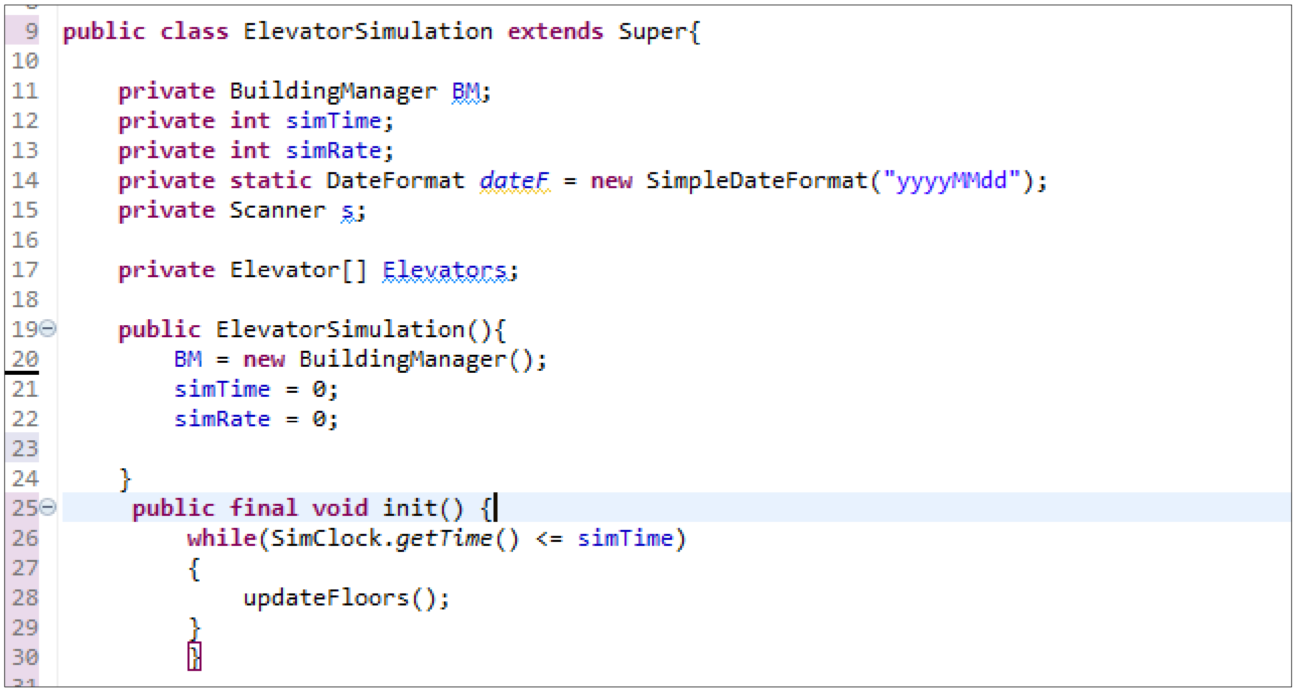
This sub section briefly discusses the first violation and how it is effecting the project flow. In the project there is an ElevatorSimulation Class that extends a Super Class. In the super class there is a constructor which has a call to a non-final method (Figure 26).



```
public class Super {
    public Super() {
    init();
    }
    public void init() {
    }
    }
```

Figure 5.1: Super Class Constructor calling a non-final method

In the subclass ElevatorSimulation, there is a constructor in which values are being initiated and an overridden init class. Creating an object of ElevatorSimulation Class to initiate values of BM, simTime and simRate, will cause a NullPointException. This is because the sub class i.e. ElevatorSimulation will implicitly call the constructor of the super class i.e. Super. This will in return call the overridden init method in the sub Class. This overridden method is using the the value of simTime without initializing its value in the subclass constructor. Hence a null pointer Exception is generated.

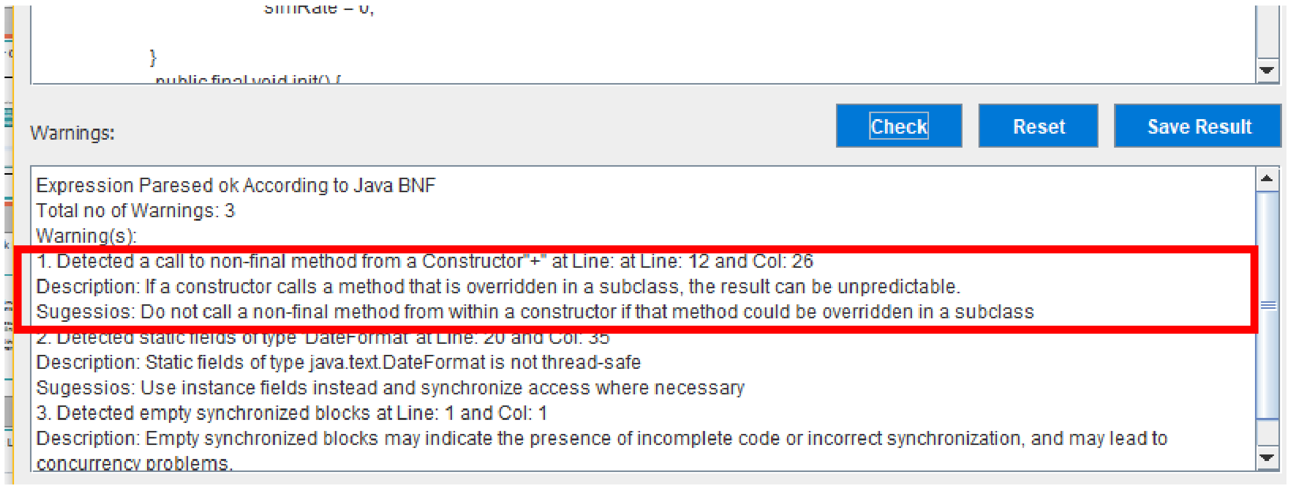


```
public class ElevatorSimulation extends Super{

    private BuildingManager BM;
    private int simTime;
    private int simRate;
    private static DateFormat dateF = new SimpleDateFormat("yyyyMMdd");
    private Scanner s;

    private Elevator[] Elevators;

    public ElevatorSimulation(){
        BM = new BuildingManager();
        simTime = 0;
        simRate = 0;

    }
     public final void init() {
         while(SimClock.getTime() <= simTime)
         {
             updateFloors();
         }
         }
```

Figure 5.2: Sub Class ElevatorSimulation

Detection Results

Our framework detects all three induced warning in the project. It also displays the location of warning and possible suggestion to fix the warning. The example warning discusses in previous sub-section is shown in figure 28.



```
Expression Paresed ok According to Java BNF
Total no of Warnings: 3
Warning(s):
1. Detected a call to non-final method from a Constructor"+" at Line: at Line: 12 and Col: 26
Description: If a constructor calls a method that is overridden in a subclass, the result can be unpredictable.
Sugessios: Do not call a non-final method from within a constructor if that method could be overridden in a subclass
2. Detected static fields of type DateFormat at Line: 20 and Col: 35
Description: Static fields of type java.text.DateFormat is not thread-safe
Sugessios: Use instance fields instead and synchronize access where necessary
3. Detected empty synchronized blocks at Line: 1 and Col: 1
Description: Empty synchronized blocks may indicate the presence of incomplete code or incorrect synchronization, and may lead to
concurrency problems.
```

Figure 5.3: Elevator Control System Results.

1.5.2. Test Project 2: Bank Customer Multi-Threading Project

Description of Project:

The Bank Customer Project allows customers to contact multiple banks and apply for loan. The bank can approve or reject these applications based on the rules set by the bank. Once all the customers get the required loan the program execution stops. However if the banks are out of funds, customers cannot get their required loan. All information regarding bank transaction, the remaining amount and the amount given to each user is displayed on the Application.

Description of Warnings Induced:

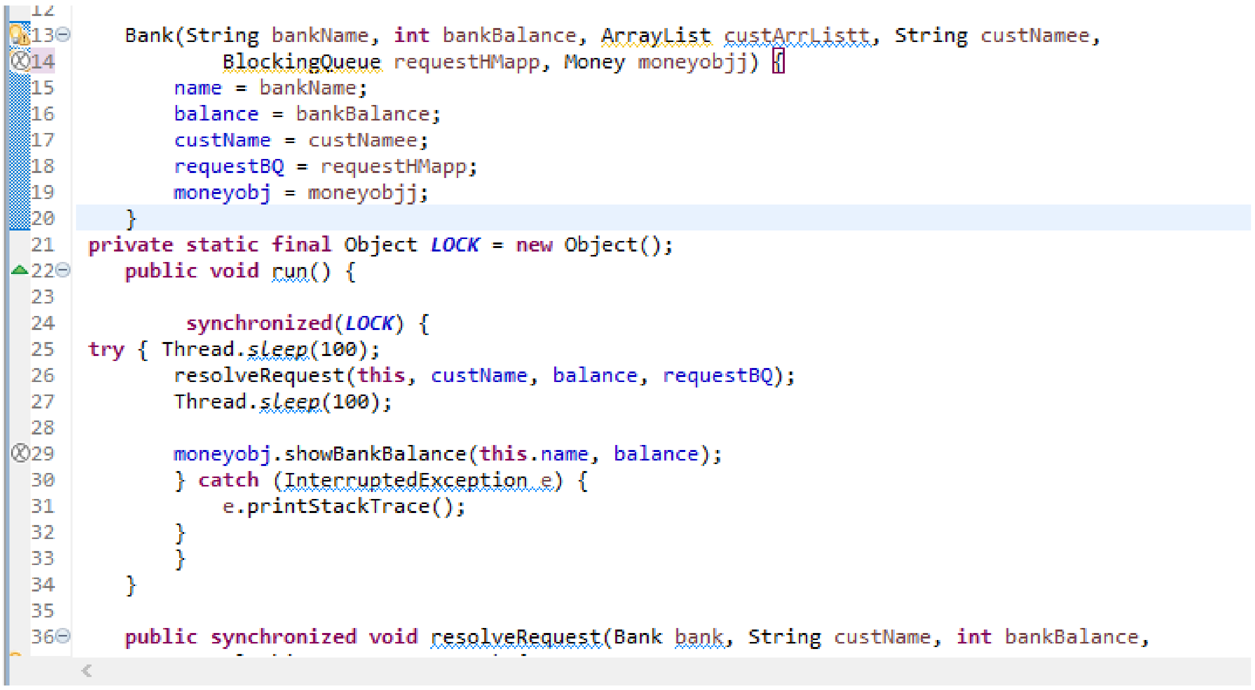
We have induced four warnings in this project based on the semantics of the code. Two of the violations are distinct and one is repeated twice. The violations are:

1) Call to thread.sleep while a lock is held

2) Setting thread priority

3) Calling wait on a Condition interface

This sub section briefly discusses the first violation and how it is effecting the program. The rest of the violations can be looked up at [63]. In this problem the thread locks object and goes to sleep. This prevents other thread from locking object. Any other thread will have to wait till this thread wakes up and unlock the object before it can continue.

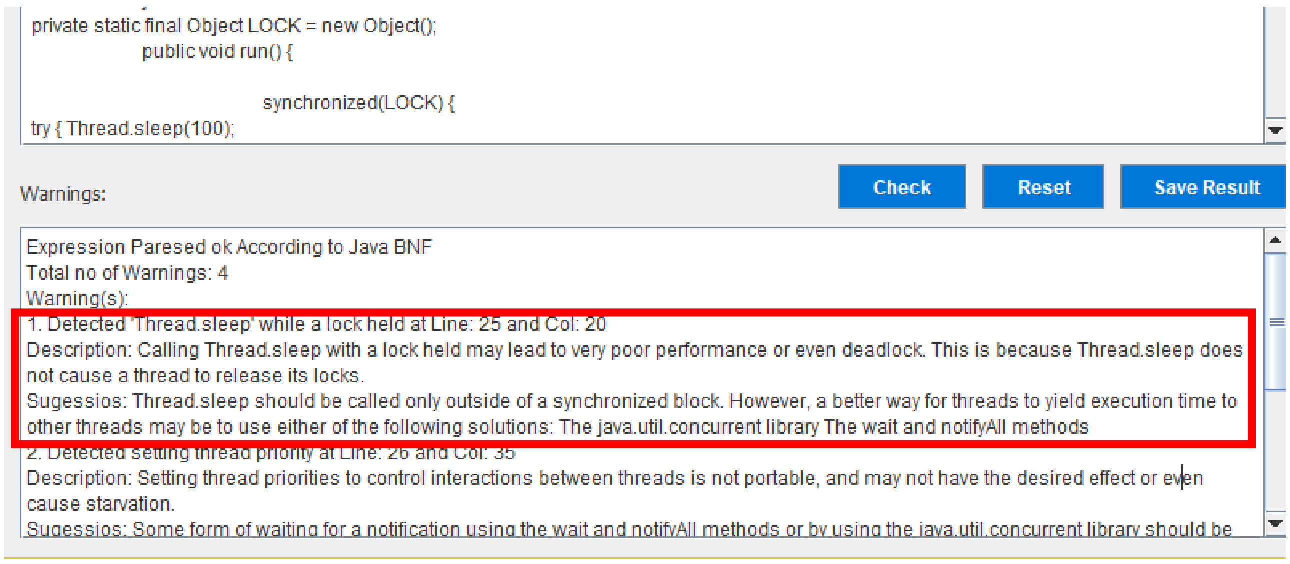


```
13      Bank(String bankName, int bankBalance, ArrayList custArrListt, String custNamee,
14              BlockingQueue requestHMapp, Money moneyobjj) {
15          name = bankName;
16          balance = bankBalance;
17          custName = custNamee;
18          requestBQ = requestHMapp;
19          moneyobj = moneyobjj;
20      }
21  private static final Object LOCK = new Object();
22      public void run() {
23
24           synchronized(LOCK) {
25  try { Thread.sleep(100);
26          resolveRequest(this, custName, balance, requestBQ);
27          Thread.sleep(100);
28
29          moneyobj.showBankBalance(this.name, balance);
30          } catch (InterruptedException e) {
31              e.printStackTrace();
32          }
33          }
34      }
35
36      public synchronized void resolveRequest(Bank bank, String custName, int bankBalance,
```

Figure 5.4: Thread sleep while a lock is held

Detection Results

All four violations are successfully detected by the framework. The detection result of the violation discussed in previous sub section is shown figure 30. The result shows the detection of warning at line 25 and column 20. It also shows how to fix the problem A possible fix is to call thread.sleep outside the synchronized block.


```
private static final Object LOCK = new Object();
                public void run() {

                                synchronized(LOCK) {
try { Thread.sleep(100);
```

Warnings:

```
Expression Paresed ok According to Java BNF
Total no of Warnings: 4
Warning(s):
1. Detected 'Thread.sleep' while a lock held at Line: 25 and Col: 20
Description: Calling Thread.sleep with a lock held may lead to very poor performance or even deadlock. This is because Thread.sleep does
not cause a thread to release its locks.
Sugessios: Thread.sleep should be called only outside of a synchronized block. However, a better way for threads to yield execution time to
other threads may be to use either of the following solutions: The java.util.concurrent library The wait and notifyAll methods
2. Detected setting thread priority at Line: 26 and Col: 35
Description: Setting thread priorities to control interactions between threads is not portable, and may not have the desired effect or even
cause starvation.
Sugessios: Some form of waiting for a notification using the wait and notifyAll methods or by using the java.util.concurrent library should be
```

Figure 5.5: Detection result of calling thread sleep with a lock held

1.5.3. Test Project 3: Multithreaded Client/Server Chat System

Description of Project:

It is a console based chat Server which uses multi-threading concepts and Java Socket Programing. The Server is open for connection with clients across the Network as well as the same Machine. Using an IP Address or Port Number the Clients can connect to the Server. Once the client is connected to the server, it can choose a unique username, join chat room, broadcast message or send/receive private messages. Java object serialization is used to transfer the messages.

Description of Error Induced:

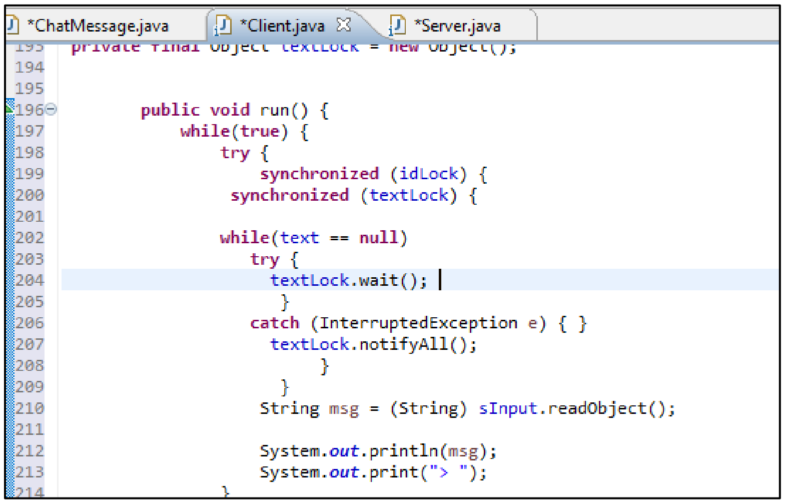
In this project we have induced four warnings in this project based on the semantics of the code. Two of the violations are distinct and one is repeated twice. The violations are:

1) Calling object.wait while tow locks are held.

2) Using notify.

3) Calling Thread.yield

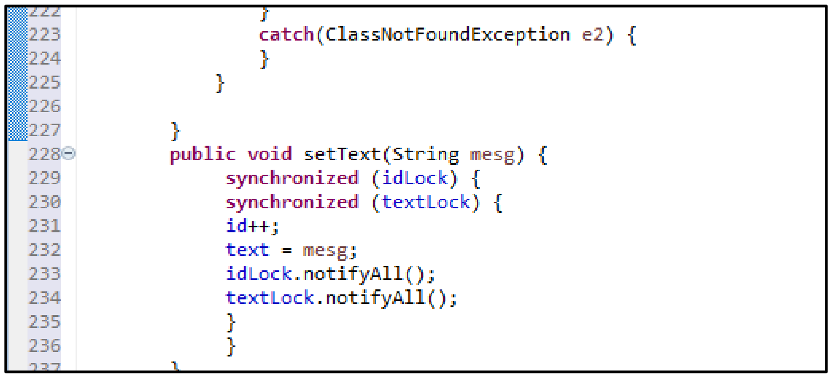
This sub section briefly discusses the first violation and how it is effecting the program. The rest of the violations can be looked up at [63]. In this problems, both idLock and textLock are locked before the value of text is read. It then calls textLock.wait, which releases the lock on textLock.



```
193     private final Object textLock = new Object();
194
195
196         public void run() {
197             while(true) {
198                 try {
199                     synchronized (idLock) {
200                  synchronized (textLock) {
201
202                 while(text == null)
203                     try {
204                       textLock.wait();
205                       }
206                     catch (InterruptedException e) { }
207                       textLock.notifyAll();
208                             }
209                       }
210                     String msg = (String) sInput.readObject();
211
212                     System.out.println(msg);
213                     System.out.print("> ");
214                 }
```

Figure 5.6: Calling object.wait while two locks are held example.

setText mothid shown in figure 32 needs to lock idLock but it cannot because idLock is still locked by run. This causes a deadlock.

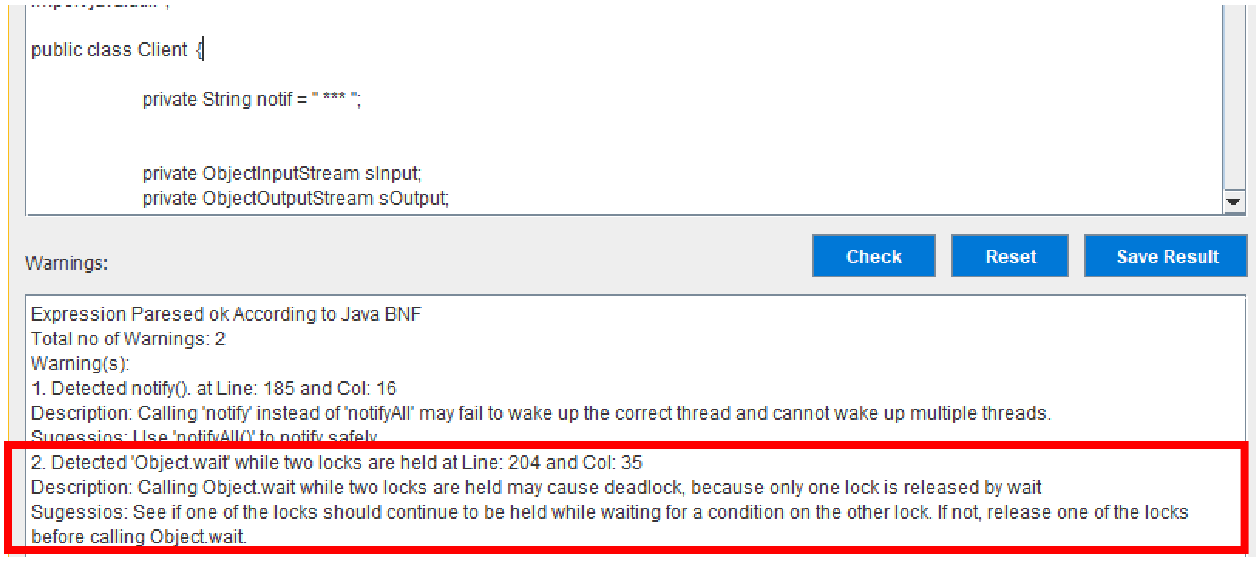


```
                }
                catch(ClassNotFoundException e2) {
                }
            }

        }
        public void setText(String mesg) {
            synchronized (idLock) {
            synchronized (textLock) {
            id++;
            text = mesg;
            idLock.notifyAll();
            textLock.notifyAll();
            }
            }
        }
```

Figure 5.7: SetText Method

Detection Results

All four violations are successfully detected by the framework. The detection result of the violation discussed in previous sub section is shown figure 33. The result shows the detection of warning at line 204 and column 35. It also shows how to fix the problem A possible fix is to release one of the locks before calling object.wait.



```
public class Client {

        private String notif = " *** ";


        private ObjectInputStream sInput;
        private ObjectOutputStream sOutput;
```

```
Expression Paresed ok According to Java BNF
Total no of Warnings: 2
Warning(s):
1. Detected notify(). at Line: 185 and Col: 16
Description: Calling 'notify' instead of 'notifyAll' may fail to wake up the correct thread and cannot wake up multiple threads.
Sugessios: Use 'notifyAll()' to notify safely
2. Detected 'Object.wait' while two locks are held at Line: 204 and Col: 35
Description: Calling Object.wait while two locks are held may cause deadlock, because only one lock is released by wait
Sugessios: See if one of the locks should continue to be held while waiting for a condition on the other lock. If not, release one of the locks
before calling Object.wait.
```

Figure 5.8: Warning result of calling object.wait while two locks are held

5.3. Standard Projects Result

We have created a dataset of Concurrency and Logical Errors in Mission Critical Systems using the analysis results from our framework. The results are given in Table 27. The term **Rule no** in table header represents the serial number of our selected rule in Sub Section 3.3. **Rule Name** represents the name of the Rule that is being violated in the standard project. Our tool detects the same rule "Do not call a non-final method from a constructor" in two projects.

Table 5.3: Result of Standard Project

SR#	Standard Project Name	Rule no	Rule Name
1	Tele Health Care System	Null	No Violation detected
2	Autonomous Driving	2.2	Do not call a non-final method from a constructor
3	Flight Navigation	2.2	Do not call a non-final method from a constructor
4	Online Analytical Processing (OLAP) Server	Null	No Violation detected
5	Automobile Cruise Control System	Null	No Violation detected

To validate our claim that, our tool improves overall reliability of the software, we have calculated the reliability of the test projects before and after it is analyzed by our tool. Results show that analyzing the test projects using our framework and using the suggested fixes significantly improves the reliability of each of the projects. Software reliability can be measures in terms of Mean Time between Failures (MTBF), Availability and Failure Rate of a Software.

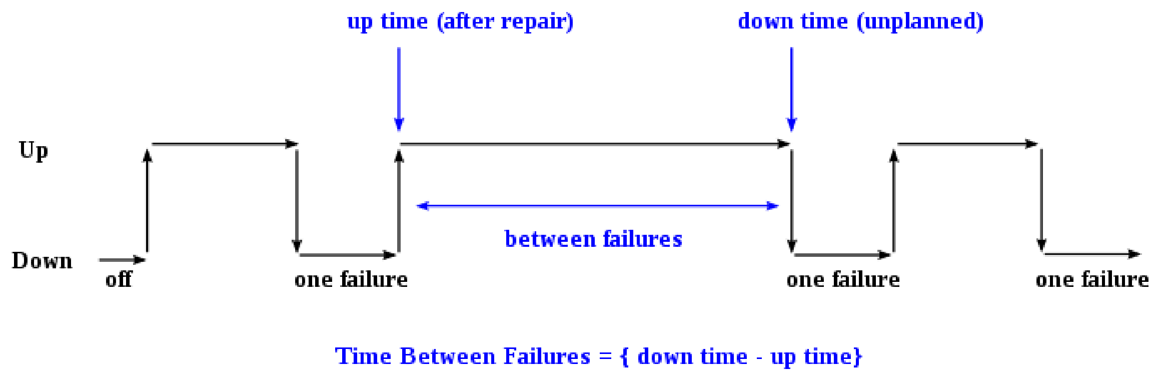


Figure 5.9: Mean Time Between Failure (MTBF)

Table 28 presents the reliability results of Standard projects. **MTBF** is the average time between consecutive failures. It is calculated using: MTBF= MTTF+MMTR. Where Mean Time to Failure (MTTF) is the average time between two consecutive failures in a software whereas Mean Time to Repair (MTTR) is the average to repair failure in a software.

MTTF is calculates using the following formula:

$$MTTF = \frac{\sum(start\ of\ downtime - start\ of\ uptime)}{number\ of\ Failures}$$

Software Availability measures the degree of software to be in operable or available state. It is calculated using the following formula:

$$Availability = \frac{MTBF}{MTBF+MTTR} \quad \text{Or,} \quad Availability = \frac{MTTF}{MTBF}$$

Failure Rate (λ) is the frequency with which a software fails. It is calculated using the following formula.

$$\lambda = \frac{1}{MTTF}$$

B represent the results before the test project is analyzed using our framework and **A** represents the results after the test project is analyzed using our framework. Delta (**Δ**) represents the difference between A and B. Negative delta values of MTBF and availability show an increase in these parameters which implies that the test project is giving failure after more time and is more available after Code Analysis using our framework. The positive delta values in Failure Rate indicates that the failure rate has decreased after code analysis of the projects using our framework.

Table 5.4: Reliability Results for Standard Projects

SR #	Error Induced Project	MTBF (MTTF+MTTR)			Availability (MTTF/MTBR)			Failure Rate λ = 1/MTTF		
		B	**A**	**Δ**	**B**	**A**	**Δ**	**B**	**A**	**Δ**
1	Tele Health Care System	206	205.5	0.0	0.91	0.91	0.00	0.0053	0.0053	0.00
2	Autonomous Driving	153.5	192.6	-39.1	0.83	0.89	-0.06	0.0077	0.0058	0.0019
3	Flight Navigation	250.3	365.5	-115.2	0.90	0.93	-0.03	0.0044	0.0029	0.0015
4	OLAP Server	166	166	0.00	0.85	0.85	0.00	0.0070	0.0070	0.00
5	Automobile Cruise Control System	207.9	207.9	0.00	0.90	0.90	0.00	0.0053	0.0053	0.00

5.4. Projects with Error Induced Result

To further validate the system we have taken 7 projects and intelligently induced some errors in it to check if our framework detects those errors. The analysis results are shown in Table 29. Our tool successfully detects all the induced errors.

Table 5.5: Analysis Results of Projects with error induced.

SR#	Project	Rule #	Rule Name	Result
1	Flight Control System	2.1 2.2	Avoid Array downcast, Do not call non-final method from constructor	Successfully detected 2 warnings
2	Bank Customer Multi-Threaded Project	1.1.1 1.4.2 1.4.3	Avoid setting thread priorities Avoid calling thread.sleep with a lock held Avoid calling wait on condition interface	Successfully detected 3 warnings
3	Chat server	1.1.2 1.1.3 1.4.1	Avoid using notify Do not call thread.Yeield Avoid calling object.wait with two locks	Successfully detected 3 warnings
4	Parent monitor	1.1.4 1.3.2 1.3.1	Do not start a thread in a constructor Ensure that a method releases lock on exit Avoid static fields of type 'DateFormat'	Successfully detected 3 warnings

5	Elastic Cloud Computing	1.1.4 1.2.1	Do not start a thread in a constructor Avoid empty synchronized blocks	Successfully detected 2 warnings
6	Hadoop distributed processing Project	2.1 2.2	Avoid Array downcast Do not call non-final method from constructor	Successfully detected 2 warnings
7	Elevator Control System	2.2 1.3.1	Do not call non-final method from constructor Avoid Static field of type dateFormat	Successfully detected 2 warnings

We have calculated the reliability of the test projects before and after it is analyzed by our tool. Results show that analyzing the test projects using our framework and using the suggested fixes significantly improves the reliability of each of the projects (Figure 30). Software reliability can be measured in terms of Mean Time between Failures (MTBF), Availability and Failure Rate of a Software. Each of the term is explained in the previous sub section. B represent the results before the test project is analyzed using our framework and **A** represents the results after the test project is analyzed using our framework. Delta (**Δ**) represents the difference between A and B. Negative delta values of MTBF and availability show an increase in these parameters which implies that the test project is giving failure after more time and is more available after Code Analysis using our framework. The positive delta values in Failure Rate indicates that the failure rate has decreased after code analysis of the projects using our framework.

Table 5.6: Reliability Results of Projects with errors induced

SR #	**Error Induced Project**	**MTBF (MTTF+MTTR)**			**Availability (MTTF/MTBR)**			**Failure Rate λ = 1/MTTF**		
		B	**A**	**Δ**	**B**	**A**	**Δ**	**B**	**A**	**Δ**
1	Flight Control System	290	553.5	-263.5	0.89	0.93	-0.04	0.0038	0.0019	0.0019
2	Bank Customer Multi-Threaded Project	266	979	-713	0.88	0.95	-0.07	0.0043	0.0011	0.0032
3	Chat server	381.4	757.5	-376.1	0.93	0.97	-0.04	0.0028	0.0013	0.0015
4	Parent monitor	178.4	400	-222.6	0.84	0.94	-0.10	0.0066	0.0026	0.0040
5	Elastic Cloud Computing	238.3	652	-414	0.87	0.95	-0.07	0.0048	0.0016	0.0032
6	Hadoop distributed processing Project	186.3	357	-170.7	0.88	0.90	-0.02	0.0061	0.0030	0.0031
7	Elevator Control System	218	413	-195	0.86	0.91	-0.05	0.0052	0.0026	0.0026

5.5. Threat to Validity

Since software is intangible, measuring software parameters is very difficult. To find reliability of our proposed framework, we have not done exhaustive testing and considered only a few types of failures other than the known failures that we have induced because we are only interested in finding the difference of reliability before and after analysis using our framework. Therefore detecting all the failures types present in the test projects has no effect on our area of interest i.e. difference in reliability. Some of the parameters that do have an effect on the validity of our reliability results are given below:

- Reliability is a customer oriented software measurement and is ideally done by getting failure reports from customer over a long period of time. For predicting reliability before release, failures can be estimated during software testing. Due to time constraint we have done the intensified stress testing and overall feature testing of each project for around an hour and reported failures after minutes.
- Not all failures are equally critical and have equal impact on reliability.
- Operation of the same project on computer with different speed can have different results.

Chapter 6

Discussion and Limitation

CHAPTER 6: DISCUSSION AND LIMITATIONS

The sub **Section 0** contains a detail discussion on proposed research work and sub **Section 0** deals with the limitation of the research.

6.1. Discussion

From this research it has been analyzed that Today's real-time systems are vastly different from traditional application programs such as Microsoft Office or AutoCAD. Modern systems such as a Nuclear Reactor Safety System is highly critical for public safety worldwide because the malfunction of a nuclear reactor can cause a serious disaster. Proper Code Analysis for Mission Critical System is very important but most of the research found on lack a framework for detecting concurrency and logical errors in MCS. Furthermore, there is no tool available in literature or Industry that checks NASA's JPL rules which are very important standards for Mission Critical Systems in Java language.

Our proposed system ensures software reliability of MCS by analyzing code against NASA JPL coding standards related to concurrency and Logical Errors since most of the errors in real time MCS occurs due to these causes. Concurrency rules are further divided into API Misuse, Synchronization, Thread Safety and Safety. We have implemented four, one, two and three rules respectively, from each subcategory. This makes a total of ten rules in concurrency category and two rules from logical error category.

Our framework extends Standard Java EBNF with our NASA's rules implementation. Based on this extended grammar a parser is generated using the JavaCC Tool. The parser we have used for parsing the code is JavaCC because not is it the most popular Java Parser but it has benefits over other important parsers such as the YACC. The benefits include top down parsing due to which more general grammar can be used, it is easier to debug and it has the ability to parse down to any non-terminal in the grammar and pass values both up and down the parse tree. Besides, it is easily customizable and has by far the largest user community and tool support. Our framework performs Hybrid Code Analysis by combining two most important code analysis techniques, the syntactic analysis and Control Flow analysis. In order to validate our proposed framework we have selected 12 open source projects for analysis of code. Out of the 12 projects we have induced violation of rules in 7 projects. Our framework successfully detects those violation and also pinpoints the location of those violations.

6.2. Limitations

This approach improves software reliability of Mission Critical System by ensuring code compliance to NASA's standards. We have implemented a small set of important rules for Mission Critical Systems. To further improve the software reliability of MCS, more of NASA's rules can be selected and implemented. Furthermore language support for languages other than java, for which standards are provided but not implemented so far, can be added into this framework. Our framework is flexible and can easily be extended with more rules and languages.

Currently, the framework is validated with some open source MCS found on Github and Source Forge. Mostly MCS, such as the systems implemented in NASA and other critical organizations, tend to keep their source code private. If our framework is validated with one or more of these systems, it can further give insights into the strength sand weaknesses of our framework.

Chapter 7

Conclusion and Future Work

CHAPTER 7: CONCLUSION AND FUTURE WORK

Our proposed framework provides a solution to improve software reliability of MCS and detect potential problems in the code early in the SDLC which can save high maintenance cost at later stages. It implements JPL Coding standards proposed by NASA to reduce chances of failure in Mission Critical Systems. To achieve this, our framework uses a hybrid technique of code analysis, combining syntactic analysis and flow analysis to detect potential problems in the code. JavaCC tool is used for parsing of code checking code conformance to our extended grammar rules.

Our approach supports detection of violation of Java Coding rules related to Concurrency and Logical errors in the code. Not only does our framework successfully detects the violations but also pin points the location of the possible cause of error in the code. It also gives a suggestion to fix the cause of failure or error as per NASA's suggestion list for each violation. The framework is successfully validated using 12 open source java MCS code.

Future work includes implementing other NASA's standards e.g. standards related to Arithmetic, Extensibility, Inefficient code etc. The framework can be extended to add support for other that have a set of coding standards which are not implemented so far, can be added into this framework.

REFERENCES

1. H. Prähofer, F. Angerer, R. Ramler and F. Grillenberger, "Static Code Analysis of IEC 61131-3 Programs: Comprehensive Tool Support and Experiences from Large-Scale Industrial Application," in *IEEE Transactions on Industrial Informatics*, vol. 13, no. 1, pp. 37-47, Feb. 2017. doi: 10.1109/TII.2016.2604760
2. I. Medeiros, N. Neves and M. Correia, "Detecting and Removing Web Application Vulnerabilities with Static Analysis and Data Mining," in *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 54-69, March 2016. doi: 10.1109/TR.2015.2457411
3. S. A. Musavi and M. Kharrazi, "Back to Static Analysis for Kernel-Level Rootkit Detection," in *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 9, pp. 1465-1476, Sept. 2014. doi: 10.1109/TIFS.2014.2337256
4. B. M. Padmanabhuni and H. B. K. Tan, "Auditing buffer overflow vulnerabilities using hybrid static–dynamic analysis," in *IET Software*, vol. 10, no. 2, pp. 54-61, 4 2016. doi: 10.1049/iet-sen.2014.0185
5. A. Bartel, J. Klein, M. Monperrus and Y. Le Traon, "Static Analysis for Extracting Permission Checks of a Large Scale Framework: The Challenges and Solutions for Analyzing Android," in *IEEE Transactions on Software Engineering*, vol. 40, no. 6, pp. 617-632, 1 June 2014. doi: 10.1109/TSE.2014.2322867
6. O. Tripp , Pietro Ferrara, and Marco Pistoia. "Hybrid security analysis of web javascript code via dynamic partial evaluation." In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 49-59. 2014
7. G. Upadhyaya and H. Rajan, "On Accelerating Source Code Analysis at Massive Scale," in *IEEE Transactions on Software Engineering*, vol. 44, no. 7, pp. 669-688, 1 July 2018. doi: 10.1109/TSE.2018.2828848
8. Y. Takhma, T. Rachid, H. Harroud, M. R. Abid and N. Assem, "Third-party source code compliance using early static code analysis," *2015 International Conference on Collaboration Technologies and Systems (CTS)*, Atlanta, GA, 2015, pp. 132-139. doi: 10.1109/CTS.2015.7210413
9. M. Thakur, and N. V. Krishna. "PYE: A Framework for Precise-Yet-Efficient Just-In-Time Analyses for Java Programs." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 41, no. 3, pp 1-37, 2019.
10. E. Sultanow, A. Ullrich, S. Konopik and G. Vladova, "Machine Learning based Static Code Analysis for Software Quality Assurance," *2018 Thirteenth International Conference on Digital Information Management (ICDIM)*, Berlin, Germany, 2018, pp. 156-161. doi: 10.1109/ICDIM.2018.8847079

11. K. P. Subedi, D. R. Budhathoki and D. Dasgupta, "Forensic Analysis of Ransomware Families Using Static and Dynamic Analysis," *2018 IEEE Security and Privacy Workshops (SPW)*, San Francisco, CA, 2018, pp. 180-185. doi: 10.1109/SPW.2018.00033
12. Y. Guo, L. Yang, X. Gao and K. Wu, "The static detection analysis technology of Android source codes," *2016 IEEE International Conference on Network Infrastructure and Digital Content (IC-NIDC)*, Beijing, 2016, pp. 288-292. doi: 10.1109/ICNIDC.2016.7974582
13. L. Lampropoulos, H. Michael, and C. Benjamin. "Coverage guided, property based testing." *Proceedings of the ACM on Programming Languages* 3, pp 1-29, 2019.
14. Z. Chengyu, N. Dellarocca, N. Andronio, S. Zanero, and F. Maggi. "Greateatlon: Fast, static detection of mobile ransomware." In *International Conference on Security and Privacy in Communication Systems*, pp. 617-636. Springer, Cham, 2016.
15. S. Khatiwada, M. Kelly and A. Mahmoud, "STAC: A tool for Static Textual Analysis of Code," *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, Austin, TX, 2016, pp. 1-3. doi: 10.1109/ICPC.2016.7503746
16. C. Nagy and A. Cleve, "A Static Code Smell Detector for SQL Queries Embedded in Java Code," *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Shanghai, 2017, pp. 147-152. doi: 10.1109/SCAM.2017.19
17. J. Ye, C. Zhang, L. Ma, H. Yu and J. Zhao, "Efficient and Precise Dynamic Slicing for Client-Side JavaScript Programs," *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Suita, 2016, pp. 449-459.
18. S. A. Mokhov, J. Paquet, and M. Debbabi. "The use of NLP techniques in static code analysis to detect weaknesses and vulnerabilities." In *Canadian Conference on Artificial Intelligence*, pp. 326-332. Springer, Cham, 2014.
19. A. Costin, "Lua Code: Security Overview and Practical Approaches to Static Analysis," *2017 IEEE Security and Privacy Workshops (SPW)*, San Jose, CA, 2017, pp. 132-142.
doi: 10.1109/SPW.2017.38
20. R. Haas, R. Niedermayr, T. Röhm and S. Apel, "Recommending Unnecessary Source Code Based on Static Analysis," *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, Montreal, QC, Canada, 2019, pp. 274-275.
doi: 10.1109/ICSE-Companion.2019.00111
21. T. T. Nguyen, P. Maleehuan, T. Aoki, T. Tomita and I. Yamada, "Reducing False Positives of Static Analysis for SEI CERT C Coding Standard," *2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry (CESI) and 6th International Workshop on Software Engineering Research and Industrial Practice*

(SER&IP), Montreal, QC, Canada, 2019, pp. 41-48. doi: 10.1109/CESSER-IP.2019.00015

22. T. Atzenhofer and R. Plösch, "Automatically Adding Missing Libraries to Java Projects to Foster Better Results from Static Analysis," *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Shanghai, 2017, pp. 141-146.
doi: 10.1109/SCAM.2017.10
23. A. Gerasimov and L. Kruglov, "Reachability confirmation of statically detected defects using dynamic analysis," *2017 Computer Science and Information Technologies (CSIT)*, Yerevan, 2017, pp. 60-64.
doi: 10.1109/CSITechnol.2017.8312141
24. G. Horváth, P. Szécsi, Z. Gera, D. Krupp and N. Pataki, "[Engineering Paper] Challenges of Implementing Cross Translation Unit Analysis in Clang Static Analyzer," *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Madrid, 2018, pp. 171-176.
doi: 10.1109/SCAM.2018.00027
25. X. Yan and H. Ma, "A New Static Vulnerabilities Analysis Algorithm for PHP Codes," *2017 International Conference on Network and Information Systems for Computers (ICNISC)*, Shanghai, China, 2017, pp. 122-125.
doi: 10.1109/ICNISC.2017.00034
26. Z. Xu and G. Liu, "STACKEEPER: A Static Source Code Analyzer to Detect Stack-based Uninitialized Use Vulnerabilities," *2018 IEEE 4th International Conference on Computer and Communications (ICCC)*, Chengdu, China, 2018, pp. 2180-2184.
doi: 10.1109/CompComm.2018.8780675
27. S. Zhao, L. Xiaohong, X. Guangquan, Z. Lei, and F. Zhiyong. "Attack tree based android malware detection with hybrid analysis." In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, pp. 380-387. IEEE, 2014.
28. Z. Ning and F. Zhang, "DexLego: Reassembleable Bytecode Extraction for Aiding Static Analysis," *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Luxembourg City, 2018, pp. 690-701.
doi: 10.1109/DSN.2018.00075
29. S. Panichella, V. Arnaoudova, M. Di Penta and G. Antoniol, "Would static analysis tools help developers with code reviews?," *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Montreal, QC, 2015, pp. 161-170.
doi: 10.1109/SANER.2015.7081826
30. B. Shastry, L. MarkuS, F. Tobias, T. Kashyap, Y. Fabian, R. Konrad, S. Stefan, S. Jean-Pierre, and F. Anja. "Static program analysis as a fuzzing aid." In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pp. 26-47. Springer, Cham, 2017.

31. R. Ramler, M. Moser and J. Pichler, "Automated Static Analysis of Unit Test Code," *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Suita, 2016, pp. 25-28. doi: 10.1109/SANER.2016.102
32. F. Cheirdari and G. Karabatis, "Analyzing False Positive Source Code Vulnerabilities Using Static Analysis Tools," *2018 IEEE International Conference on Big Data (Big Data)*, Seattle, WA, USA, 2018, pp. 4782-4788. doi: 10.1109/BigData.2018.8622456
33. Kurniawan, A., Abbas, B.S., Trisetyarso, A. and Isa, S.M. "Static Taint Analysis Traversal with Object Oriented Component for Web File Injection Vulnerability Pattern Detection." *Procedia Computer Science* 135, 2018, pp 596-605.
34. A. Muhammad, T. A. Tanveer, M. Tehseen, M. Khan, F. A. Khan, and S. Anwar. "Static malware detection and attribution in android byte-code through an end-to-end deep system." *Future Generation Computer Systems*, 2020, pp 112-126.
35. O. Javed, Z. Yudi, R. Andrea, S. Haiyang and B. Walter. "Extended code coverage for AspectJ-based runtime verification tools." In *International Conference on Runtime Verification*, pp. 219-234. Springer, Cham, 2016.
36. S Blazy, D Bühler, B Yakobowski "Improving static analyses of C programs with conditional predicates." *Science of Computer Programming* 2016 pp 77-95
37. W Niu, X Zhang, X Du, L Zhao, R Cao, M Guizani "A Deep Learning Based Static Taint Analysis Approach for IoT Software Vulnerability Location." *Measurement,* 2019.
38. J Song, C Han, K Wang, J Zhao, R Ranjan, "An integrated static detection and analysis framework for android." *Pervasive and Mobile Computing* 32, 2016, pp 15-25.
39. C. Huang, C. Chiu, C. Lin and H. Tzeng, "Code Coverage Measurement for Android Dynamic Analysis Tools," *2015 IEEE International Conference on Mobile Services*, New York, NY, 2015, pp. 209-216. doi: 10.1109/MobServ.2015.38
40. A. Sakti, G. Pesant and Y. Guéhéneuc, "Instance Generator and Problem Representation to Improve Object Oriented Code Coverage," in *IEEE Transactions on Software Engineering*, vol. 41, no. 3, pp. 294-313, 1 March 2015. doi: 10.1109/TSE.2014.2363479
41. M. K. Alzaylaee, S. Y. Yerima and S. Sezer, "Improving dynamic analysis of android apps using hybrid test input generation," *2017 International Conference on Cyber Security And Protection Of Digital Services (Cyber Security)*, London, 2017, pp. 1-8. doi: 10.1109/CyberSecPODS.2017.8074845
42. Duck, G. J., & Yap, R. H. C. "*EffectiveSan: type and memory error detection using dynamically typed C/C++". Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2018*

43. Haller, I., Jeon, Y., Peng, H., Payer, M., Giuffrida, C., Bos, H., & van der Kouwe, E. "*TypeSan,* Practical Type Confusion Detection". *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16, 2016.*
44. X. Wang, Y. Zhang, L. Zhao and X. Chen, "Dead Code Detection Method Based on Program Slicing," *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, Nanjing, 2017, pp. 155-158. doi: 10.1109/CyberC.2017.69
45. M. Y Hong, L. R. Qin, "Application of dynamic program slicing technique in test data generation", *8th International Conference on Advances in Information Technology, IAIT2016*, China, December 2016, pp 19-22.
46. A. Treffer and M. Uflacker, "The Slice Navigator: Focused Debugging with Interactive Dynamic Slicing," *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Ottawa, ON, 2016, pp. 175-180.
47. D. Kang, J. Sohn, and S. Yoo. "Empirical evaluation of conditional operators in GP based fault localization". *Genetic and Evolutionary Computation Conference (GECCO '17).* ACM, New York, NY, USA, 1295-1302
48. M. Papadakis and Y. L. Traon. "Effective fault localization via mutation analysis: a selective mutation approach". *29th Annual ACM Symposium on Applied Computing (SAC '14*). ACM, New York, NY, USA, 2014, pp 1293-1300
49. J. Sohn and S. Yoo. "FLUCCS: using code and change metrics to improve fault localization". *26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA 2017), ACM, New York, NY, USA, 2017. pp 273-283.
50. G. Pellegrino, C. Tschürtz, E. Bodden and Rossow, "Using dynamic analysis to crawl and test modern web applications." *International Symposium on Recent Advances in Intrusion Detection*. Springer, Cham, 2015.
51. O. Rodriguez-Prieto, A. Mycroft and F. Ortin, "An Efficient and Scalable Platform for Java Source Code Analysis Using Overlaid Graph Representations," in *IEEE Access*, vol. 8, pp. 72239-72260, 2020,
52. S. Charters and B. Kitchenham, ''Guidelines for performing systematic literature reviews in software engineering,'' Keele Univ. Durham Univ. Joint Rep., Newcastle, U.K., Tech. Rep. EBSE-2007-01, Version 2.3, 2007.
53. J. -. Jazequel and B. Meyer, "Design by contract: the lessons of Ariane," in *Computer*, vol. 30, no. 1, pp. 129-130, Jan. 1997, doi: 10.1109/2.562936.
54. T. Taylor, G. VanDyk, L. W. Funk, R. M. Hutcheon and S. O. Schriber, "Therac 25: A New Medical Accelerator Concept," in *IEEE Transactions on Nuclear Science*, vol. 30, no. 2, pp. 1768-1771, April 1983, doi: 10.1109/TNS.1983.4332638.
55. B. David. "Source code analysis: A road map." In *Future of Software Engineering (FOSE'07)*, pp. 104-119. IEEE, 2007.
56. PMD.[Online], https://pmd.github.io, accessed on May 2021.
57. Findbugs.[Online], http://findbugs.sourceforge.net, accessed on May 2021.

58. A. A. Kulkarni, and J. Aghav. "Automated techniques and tools for program analysis: Survey." In *2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*, pp. 1-7. IEEE, 2013.
59. I. García-Ferreira, C. Laorden, I. Santos and P. Garcia Bringas, "Static analysis: a brief survey," in *Logic Journal of the IGPL*, vol. 24, no. 6, pp. 871-882, Dec. 2016. doi: 10.1093/jigpal/jzw042
60. A. Andrei, S. Ciobâca, V Craciun, D.Gavrilut, and D. Lucanu. "A comparison of open-source static analysis tools for vulnerability detection in c/c++ code." In *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pp. 161-168. IEEE, 2017.
61. Q. A. Ain, W. H. Butt, M.W. Anwar, F.Azam, and B. Maqbool. "A systematic review on code clone detection." *IEEE Access* 7 (2019): 86121-86144.
62. Plsek, A., Zhao, L., Sahin, V.H., Tang, D., Kalibera, T. and Vitek, J., 2010, August. Developing safety critical Java applications with oSCJ/L0. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems* (pp. 95-101).
63. Java Coding Standards.[Online], https://www.havelund.com/Publications/JavaCodingStandard.pdf, accessed on March 2021.
64. Elevator Multithread-Java.[Online], https://github.com/jzeng5/Elevator-multithread-Java, accessed on May 2021.
65. Bank-Customer Multithreading-Project.[Online], https://github.com/Chetanpaliwal22/Bank-Customer-Multithreading-Project, accessed on May 2021.
66. Chat-Server.[Online], https://github.com/abhi195/Chat-Server, accessed on May 2021.
67. Autonomous Driving.[Online], https://github.com/klevis/AutonomousDriving, accessed on May 2021.
68. Flight-Control-System.[Online], https://github.com/SHRMu/Flight-Control-System-Java-RMI, accessed on May 2021.
69. Autonomous Driving.[Online], https://github.com/klevis/AutonomousDriving, accessed on May 2021.
70. Flight Control System.[Online], https://github.com/SHRMu/Flight-Control-System-Java-RMI, accessed on May 2021.
71. Mondarian.[Online], https://github.com/SHRMu/Flight-Control-System-Java-RMI, accessed on May 2021.

72. PRV.[Online]. https://github.com/srfunksensei/PRV, accessed on May 2021.
73. Tele Health Care.[Online], https://github.com/neeleshsaxena/Tele-HealthCare, accessed on May 2021.
74. Flight Conrol.[Online], https://github.com/SHRMu/Flight-Control-System-Java-RMI, accessed on May 2021.
75. Parent Monitor.[Online], https://github.com/VisionZ/Parent-Monitor, accessed on May 2021.
76. Cloud Computing Elasticity.[Online], https://github.com/Deivakumaran/Elasticity-in-Cloud-Computing-using-Multithreaded-Programming, accessed on May 2021.
77. Cloud Computing.[Online], https://github.com/absnaik810/CloudComputing/blob/master/Project%201/ahnaik_project1, accessed on May 2021.
78. Java Analyzer.[Online], https://github.com/rkaydivergent/JavaAnalyzer, accessed on May 2021.
79. Checkstyle.[Online], https://checkstyle.sourceforge.io/, accessed on May 2021.
80. Dead Code Detector.[Online], https://www.softpedia.com/get/Programming/Other-Programming-Files/DCD-Dead-Code-Detector.shtml, accessed on May 2021.
81. DepFinfder.[Online], https://depfind.sourceforge.io/, accessed on May 2021.
82. JLint.[Online], https://sourceforge.net/projects/jlint/, accessed on May 2021.
83. Sonarqube.[Online], https://www.sonarqube.org/, accessed on May 2021.
84. Facebook Infer.[Online], https://www.facebookInferorg/, accessed on May 2021.