**UNKOWN FAILURE DETECTION BY CALL STACK TREE**

APPROVED BY SUPERVISING COMMITTEE:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Dr. Junghee lee , Ph.D., Chair

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Dr. lide duan, Ph.D., Co- Chair

Accepted: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Dean, Graduate School

# DEDICATION

*I dedicate this project to UTSA- Electric Engineering Department and thank you for providing this opportunity and required resources. Also, I sincerely acknowledge the continuous help from Dr. Junghee lee in accomplishment of this project.*

**UNKNOWN FAILURE DETECTION BY CALL STACK TREE**

by

SAI CHANDRANEEL DODDA, B.Tech (EE)

GRADUATE PROJECT

Presented to the Graduate Faculty of

The University of Texas at San Antonio

in Partial Fulfillment

of the Requirements for the Degree, of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

THE UNIVERSITY OF TEXAS AT SAN ANTONIO

College of Engineering

Department of Electrical Engineering

August 2017

# ACKNOWLEDGEMENTS

I am highly indebted to my advisor Prof. Junghee Lee for this constant encouragement and motivation. This project and report would not have been this affective without his support.

I would also like to thank my graduate project committee member Prof. Lide Duan for his support and University of Texas at San Antonio for providing required resources.

**UNKOWN FAILURE DETECTION BY CALL STACK TREE**

# by

Sai Chandraneel Dodda, B.Tech (ECE)

The University of Texas at San Antonio, 2017

Supervising member: Dr. Junghee lee

**ABSTRACT:**

The purpose of this project to detect failures in the IOT devices. A GUI application is built using java which has two modes called training mode and detection mode. An input trace file is taken which builds a call stack tree when the iot device is tested which is done in training mode. In detection mode ,the call stack tree is compared by the online call stack by call stack monitor. If it matches it gives output as “pass” or else “fail”. In this project we can also discuss different types of failures that occur in the IOT devices

# TABLE OF CONTENTS

ACKNOWLEDGEMENT ............................................................................................................ iv

ABSTRACT ................................................................................................................................... v

LIST OF FIGURES .....................................................................................................................vii

Chapter 1: Introduction ....................................................................................................................1

IOT devices…………………………………………………………………………….1

Applications……………………………………………………………………………1

Chapter 2: IOT device failures .....................................................................................................4

Proposed approach ……………..................................................................................4

Proposed monitoring system ..…..................................................................................6

Hardware based monitoring agent……………………………………………………...6

Chapter 3: Unknown failure detection of call stack tree ................................................................9

Chapter 4: Limitations and future work..........................................................................................12

Chapter 5: Java classes and packages……………………………………………………………..13

Chapter 6: Overview of codes and results………………………………………………………..14

Results……………………………………………………………………………….46

Conclusion .....................................................................................................................................50

REFERENCES…………………………………………………………………………………...51

Vita ………………………………………………………………………………………………52

**LIST OF FIGURES**

Figure 1: A high-level overview of the proposed monitoring system for the examination of IoT devices……………………………………………………………………………………………..5

Figure 2: The proposed Call Stack Monitor agent incorporated within the processor to monitor the function call stack …………………………………………………………………………………6

Figure 3: The block diagram of proposed hardware-based call stack monitor …………………..9

Figure 4: An example of call stack tree.........................................................................................10

Figure 5 Pseudocode for building and comparing of call stack tree................................................11

Figure 6: GUI Displayed with training and detection mode........................................................46

Figure 7: Call stack tree is displayed ............................................................................................47

Figure 8: Output for pass .............................................................................................................48

Figure 9: Output for fail…………………………………………………………………………49

**CHAPTER ONE :INTRODUCTION**

**IOT DEVICES:**

Iot is the inter-networking of physical devices such as connected devices like buildings , software, sensor which enables them to collect data and also exchange data. It allows object to be controlled remotely which creates direct integration of physical world in network infrastructure. This results in improved accuracy and efficiency in addition to reduced human intervention. This one is used in technologies like smart grids, virtual power plants and smart cities. Each thing is known through its embedded computing system which functions within present internet infrastructure.Iot is able to offer advanced connectivity of devices and system. This includes beyond machine to machine(M2M) communication ,protocols ,domains and applications. These devices are used to generate large amounts of data from different locations

**APPLICATIONS:**

**MEDIA:**

iot devices plays a major role in media .As typical media they depending upon newspaper ,articles , TV commercials .The media process the data in dual and interconnected manner

• Target consumers

• Data-capture

This internet of things create a opportunity to measure collect analyze variety of behavioral statistics. Cross-correlation of this data could change marketing nature of products and services. By this wealth they get can allow practitioners in advertising and media to gain an elaborate layer on present mechanism used by the industry

**ENVIRONMENTAL MONITORING**:

Internet of things typically use sensors which assist in environmental protection by monitoring soil, water quality and atmospheric conditions. They are also used in monitoring wild life and natural resources. They can also predict the occurrence of tsunami and earth quake which results in massive chaos in the world and also used by emergency services to provide aid to the people .iot in wireless sensing will be ground breaking**.**

**INFRASTRUCTURE MANAGEMENT**:

Iot devices are used in monitoring and controlling operations of rural and urban infrastructures like bridges,railway tracks .They also keep track of scheduling repairing and maintenance activities in efficient manner.Iot devices are also used in controlling infrastructure costs like improve quality of service , reduce cost of equipment,deploy emergency response co-ordination.Even areas such as waste management can benefit from automation and optimization that could be brought in by Iot device

**MANUFACTURING**:

Iot devices bring in realm of industrial application and smart manufacturing.Smart Iot enavles rapid manufacturing process of new products, real time optimization of manufacturing production and supply chain.I t is used in predictive maintenance, statistical evaluation.smart systems have smart grids which provide health and safety management, automation, plant optimization etc lot of other function which is done by a large number of networked sensors

**ENERGY MANAGEMENT**:

Integration of sensing and actuation systems connected to internet which optimizes energy consumption as a whole.They integrate in all forms of energy consuming devices . which able to communicate with utility supply company in order to effectively balance power generation and energy usage. Such devices would also offer the opportunity for users to remotely control their devices.Home based energy management similar to smart grid which has different types of goals to improve efficiency , reliability , economics and sustainability of production and distribution of electricity

**MEDICAL AND HEALTHCARE**:

Iot devices can be used in health monitoring and emergency notification systems.These health monitoring devices can range can range with specialized Fitbit pacemaker. They also are able monitor when the patient is getting up from the bed by using smart beds.They are also monitoring vitals and blood pressure of senior citizens .They are also providing appropriate pressure and support on the patient without manual interaction of nurses.

**BUILDING AND HOME AUTOMATION**:

Iot devices can be used to monitor and control the mechanical ,electrical and electronic systems used int various types of buildings which may be public, private sectors

**TRANSPORTATION:**

Iot can assist in integration of communications , control and information processing across various systems.dymanic interaction with various component of transport system enables inter and intra vehicular communication ,smart traffic control, smart parking and road assistance

**CONSUMER APPLICATION:**

Iot devices has wide based consumer use like connected car, entertainment ,wearable technology, quantified self, connected health and smart retail. But some applications are so disliked by the consumers they kept on asking because they lack redundancy and their inconsistency. They started it calling “internet of shit” companies rushed into iot creating devices of no good use which has terrible security standards. Consumer iot provides new opportunities for user experience and interfaces.

**CHAPTER TWO: IOT DEVICE FAILURES**:

After iot devices are developed and tested, they are integrated with system and deployed. Due complex nature of Iot systems however they may fail due to deployment. Individual Iot devices has mechanism which can report their own problem. The issue is they are only able to report known failures that occur in devices. Self-diagnosis techniques can be employed to handle unknown failures. But low end devices can’t afford this type of techniques due to their tight resource constraints. In this situation Iot devices does not work which makes them unable to recognize or report the problem. Then first responders should look at the problem which is time consuming. An automated and generalized methodology is required for examining several heterogenous devices. Well the methodology is based on a very lightweight processor level architectural support. A hardware-based monitoring agent within processor and connected to a separate monitoring device when an examination is required. By analyzing information we can determine if the device is working or not. The monitoring agent keeps track of the function call stack.

**PROPOSED APPROACH :**

When an IoT system fails after deployment and no device reports a failure, someone (typically, a ﬁrst responder) has to examine the problem. The ﬁrst step is to examine all devices to identify the failed one. As previously mentioned, the problem with the failed device can then be classiﬁed as an operational problem (such as misconﬁguration, or user error), a device failure (the device needs to be replaced), or a design error (the device needs further debugging). Throughout this paper, we distinguish fault and failure as follows. • Fault: [Deﬁnition] • Failure: [Deﬁnition] A fault does not always lead to failure if it is properly handled, which is the purpose of existing fault-tolerance or selfdiagnosis techniques. In contrast, the purpose of the proposed examination methodology is to quickly identify failed devices, not to detect and handle faults. In a large-scale IoT system, examining a number of heterogeneous devices is not a trivial task. Here, “heterogeneous” means that the types of devices may be diﬀerent; thus, they may require diﬀerent methods to examine their correctness. If a single tool and/or method require in-depth knowledge and manual examination, it will take a long time to examine all devices. Therefore, an automated and generalized methodology that can be applied to diﬀerent types of devices is imperative for the examination of a large-scale IoT system. To realize an automated methodology for examination, a run-time monitoring system is needed. Its requirements are as follows: • Automation: An automated tool should be provided to determine whether or not the device is working correctly. • Standardization: The interface of the monitoring system should be standardized to enable the examination of multiple devices. Otherwise, ﬁrst-responders need to be equipped with many diﬀerent monitoring systems. • Non-intrusiveness: The overhead of the monitoring system should be minimized. If the monitoring system incurs non-negligible overhead, it may distort operations of realtime IoT devices. • Hot plug: The monitoring system should be able to connect to the IoT device, while the IoT device is running. Toward this end, we envision a monitoring system as illustrated in Figure 1. The examiner is able to use a light-weight



Figure 1: A high-level overview of the proposed monitoring system for the examination of IoT devices.

device – e.g., a smart phone – as an external monitoring device and take it to wherever IoT devices are installed. A smart phone is an ideal monitoring device, because modern smart phones oﬀer relatively high computing power, high mobility, and various types of connectivity. The phone is connected to an IoT device through a cable, or any kind of wireless communication. A cable (wired communication) is the best way to save power. Regardless, in an environment where wired communication is not possible, wireless communication can still be used

**PROPOSED MONITORING SYSTEM:**

This section explains the details of two main components of the proposed monitoring system: the monitoring agent (located within the IoT device), and the external monitoring program (which runs on the external monitoring device, as depicted in Figure 1).

**HARDWARE BASED MONITORING AGENT:**

The most critical question related to the development of the monitoring system is: What should be monitored? Software debuggers monitor all internal state of a processor, including the program counter, the registers, and the memory contents. They are not, however, monitored in real-time, since the debugger must stop (with a break point) to show this state. Hardware debuggers often include hardware signals, as well as traces of software execution. Since these debuggers generate a large amount of trace data, they can either work for a short period of time, or stop regularly,



Figure 2: The proposed Call Stack Monitor agent incorporated within the processor to monitor the function call stack. [Distinguish between the NEW component and the existing one (transmitter).]

in order to avoid losing trace information. Diagnostic techniques collect application-speciﬁc data to determine unexpected failures. Instruction-grained monitoring techniques monitor the architectural state of every instruction. Security audit techniques usually monitor a part of the system to detect a particular type of attack that they are targeting. For example, KI-mon [22] monitors the system bus to detect unauthorized modiﬁcations of mutable data structures of the operating system kernel. In this paper, function calls are monitored for examination. Monitoring only function calls does not incur much overhead, provides enough information for examination, and can be easily applied to diﬀerent devices. Hardware problems would eventually aﬀect the ﬁrmware, which would also be detected by monitoring the function calls. By monitoring function calls, we can detect failures caused by a processor (and/or its software), and also peripheral hardware outside the processor. It will be demonstrated that monitoring function calls is a cost-eﬀective way to detect most types of unexpected failures. Figure 2 shows a processor that is equipped with the proposed monitoring agent. The key new component is the call stack monitor, which keeps track of the function call stack. It sends notiﬁcations to the external monitoring program through the transmitter. We may reuse existing communication peripherals/protocols, e.g., Universal Serial Bus (USB) and Universal Asynchronous Receiver Transmitter (UART). To check the sanity of the IoT device, the call stack should be compared against a reference. The reference is a call stack tree, which is be explained in Section 4.2. Figure 3 shows a detailed block diagram of the call stack monitor module. Note that this is a lightweight hardwarebased structure incorporated within the processor. The call stack monitor keeps track of the call stack. When a branch instruction is executed and is determined to be taken, its branch target address is pushed into the call stack. In our implementation, the call stack is implemented with registers (not SRAM memory) to enable fast access times. A branch instruction is any instruction that changes the program counter to a target address and – at the same time – stores the return address into a link register (e.g., the bl instruction in the ARM instruction set). If a return instruction is executed, the last branch target address is popped from the stack. Here, a return instruction is any instruction that updates the program counter to the value stored in the link register. Whenever any change is made to the call stack, the monitoring program (running on the external

Figure 3: The block diagram of the proposed hardware-based call stack monitor. [Confusing: not easy to see which part is the actual hardware component (i.e., the call stack).]

monitoring device) is notiﬁed. If the monitoring program is not connected, the call stack monitor keeps track of the call stack without notiﬁcations. Once the monitoring program (i.e., the external monitoring device running the monitoring program) is connected to the IoT device, the call stack monitor sends a notiﬁcation, after it synchronizes the current snapshot of the call stack with the external monitoring program. When a branch instruction is executed, we may store its target address, or its return address. For the purpose of examination, both would work, because the proposed methodology detects failures by detecting abnormal function calls. In this paper, we decided to store the branch target address for the convenience of post-analysis. It is typically easier to identify which functions were called by observing the branch target addresses, rather than the return addresses. The call stack monitor keeps monitoring the control signals and the datapath of the processor to detect pertinent instructions. Since the call stack monitor only reads (i.e., observes) signals from the processor, it does not aﬀect the operation and/or the performance of the processor itself. However, if the call stack monitor generates and sends too many events to the external monitoring program, the event queue between the call stack monitor and the monitoring program might become full. To avoid losing these events, the processor must be stalled. In other words, as long as the external monitoring device can keep up with the execution pace, the call stack monitor does not incur any performance degradation. Note that the responsibility of the call stack monitor is only to collect function call traces and pass them to the external monitoring program. The monitoring program is in charge of analyzing the stack to determine whether or not the device is working correctly. The transmitter in the monitoring agent is in charge of sending messages to the external monitoring device. We can send messages through USB without the help of software, i.e., the hardware can autonomously send messages. This is useful, for example, in a situation when unexpected failures occur and we cannot trust the software. Autonomous hardware can ensure continuous communication between the monitoring agent and the external monitoring device.



Fig 3 :The block diagram of proposed hardware-based call stack monitor

**CHAPTER THREE: UNKNOWN FAILURE DETECTION BY CALL STACK TREE:**

A call stack tree is a collection of reachable call stack snapshots represented as a tree. The call stack tree is built when the IoT device is being tested. We will refer to the time when the call stack tree is being built as the “training” mode, and to the time when the call stack tree is being compared with the online call stack collected by the call stack monitor as the “detection” mode. In the detection mode, if the on-line call stack does not match with the call stack tree, a fault is declared.

Figure 1 shows an example of the call stack tree. If main, funcA and funcX are called in this order, from the bottom to the top. This is represented as the leftmost path of the tree (from main to the leftmost funcX). After funcX and funcA are returned, let us suppose funcB is called at a certain moment, but funcX or funcY is not yet called. At this moment, the call stack is main and funcB. It is represented by the right path of the tree (from main to funcB). Therefore, the call stack tree captures all reachable call stack snapshots in a tree form.

While the tree is being built (training mode), the length of time it takes for each function to be executed will be measured to detect hanging (deadlock or livelock). For each function, the statistics of the average and standard deviation of execution time will be calculated in the training mode. In the detection mode, if a function takes longer than the threshold, which is pre-determined based on statistics obtained in the training mode, a fault is detected. By using the call stack tree, we can detect the following three types of failure.

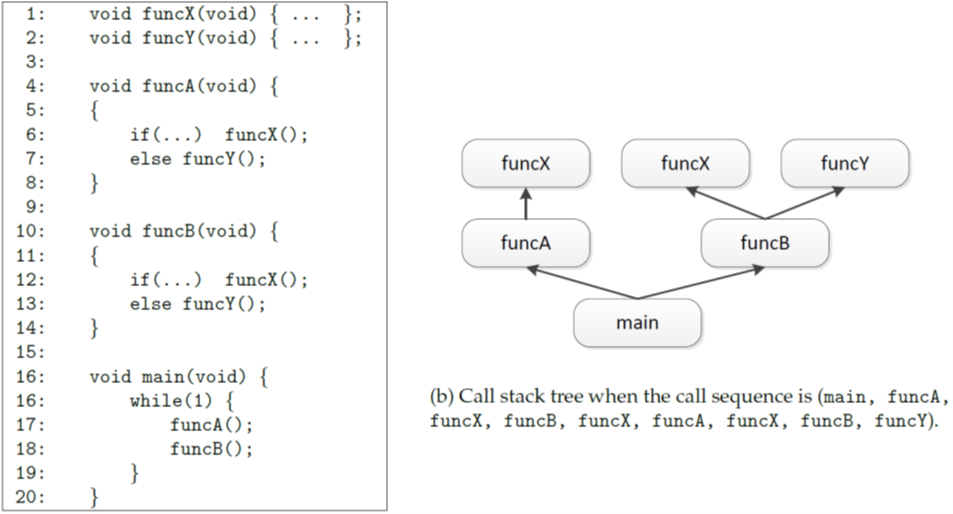


Fig 4: An example of call stack tree

• If unknown failures eventually lead to known failures: When known faults occur, their corresponding exception handlers are usually called. The IoT device may have a reporting mechanism but, if the reporting mechanism itself has a fault, the fault cannot be reported. If the call stack tree contains any of the exception handlers, it means known faults have occurred.

• If the device is hanging: If the IoT device is stuck at any point, it can be detected based on the statistics of execution time as explained above.

• If the device seemingly keeps working but incorrectly: If the on-line call stack does not match with the call stack tree, a fault can be detected.

However, it cannot detect the following type of unknown failure.

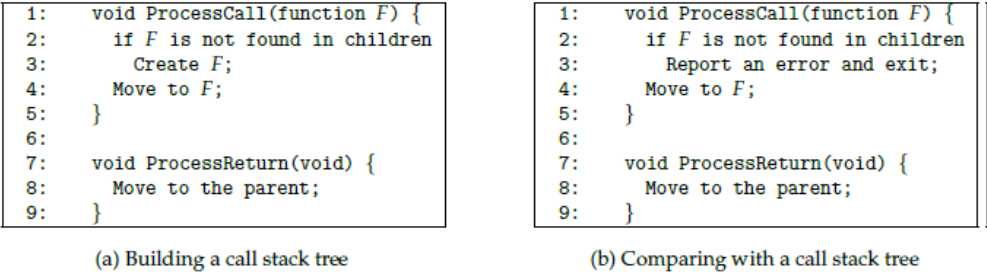
• If unknown failures affect the output, but the output is still in a legitimate form: To detect this type of failures, the output should be compared with a golden model. To address this, we will investigate an assertion-based examination methodology in our future work.

Fig 5: Pseudocode for building and comparing with a call stack tree

The call stack monitor has three operational states: monitor, activate, and notify. In the monitoring state, the call stack monitor is not connected to an external monitoring device. When the external monitoring device is connected, the status switches to the activate state, where the snapshot of the call stack is transferred to the external monitoring device. Once the transfer completes, the call stack monitor

switches to the notify state. In this state, any changes to the call stack are immediately sent to the external monitoring device. Thus, the call stack is maintained in both the call stack monitor and the external monitoring device. For example, let us go back to Figure 4. The external monitoring program has the pre-built (reference) call stack tree, as shown in Figure 4(b). The monitoring program maintains a pointer that indicates which function the IoT device is currently executing. At the beginning, the pointer is at the root. When the external monitoring device is connected, let us suppose that main and funcB are in the stack. In the activate state, they are transferred to the external monitoring device. When the main function is received, the pointer moves to main – connected with the root – in the call stack tree. When the funcB function is received, the pointer moves to funcB, which is the right child of main in the call stack tree. In this example, all functions in the call stack are found in the call stack tree in the right order. Thus, the system is considered to be working correctly so far. After sending the addresses in the stack to the external device, the call stack monitor switches to the notify state. In this state, any function calls and returns are sent to the external monitoring program. When a function call is received, the monitoring program searches for it among children nodes of the current location. If found, the pointer moves to that position and records the current time. If not found, a failure is declared. When a function return is received, the pointer moves to the parent node and the execution time is calculated. If the execution time is too short, or too long, a failure is declared. For example, let us suppose funcX is received. Since it is found among the children of the current node (funcB), the pointer moves to funcX and the current time is recorded. When a function return is received, the pointer moves to the parent (funcB), and the execution time of funcX is calculated. At this moment, let us suppose funcZ is called. Since this function cannot be found among the children of funcB, this abnormal execution ﬂow is indicative of a failure.

**CHAPTER FOUR :LIMITATIONS AND FUTURE WORK**:

If multiple processes are running on the IoT device, their call stacks should be monitored separately. The call stack monitor needs scheduling information – speciﬁcally, which process is currently running – from the operating system to keep track of the current call stack. If a non-standard control ﬂow is used (e.g., software signal handlers, or setjmp/ longjmp), or if a Dynamically Linked Library (DLL) is used , the call stack monitor will need additional information from the operating system. These issues will be addressed in our future work. One such example is assertion-based veriﬁcation. Assertion-based veriﬁcation is a popular methodology for veriﬁcation of both hardware and software. Since it automatically checks the sanity of software and is applicable to any device, it is suitable for examination of IoT devices. To reduce performance overhead, a hardware-software cooperative solution may be adopted.

**CHAPTER FIVE: JAVA PACKAGES AND CLASSES**:

Packages in Java are used to prevent naming conflicts. A Package is defined as a grouping of classes, interfaces, enumerations and annotations providing access protection and namespace management.

Packages in Java are a method for gathering comparable types of classes/interfaces together. We can essentially import a class giving the required functionality from a current package and utilize it in our program. A package may consist of a lot of classes but only a few needs to be exposed as most of them are required internally. In this manner, we can conceal the classes and prevent programs or different packages from accessing classes which are implied for internal use only.

Package in java can be categorized in two form

1. Built-in Package (Standard Packages as a part of Java Runtime Environment)

2. User-Defined Package (Packages Defined by programmers to group the related classes)

Package Name Description

**Java.lang** : Contains language support classes

**Java.io**  : Contains classes for supporting input/output operations.

**Java.net**  : Contains classes for supporting networking operations.

**Java.awt** : Contains classes for implementing GUI (Graphical User Interface)

**Java.applet**: Contains classes for creating Applets

**Java.util** : Contains classes for implementing data structures and for Date/Time Operations.

**Javax.swing:** contains classes for swing API such as Jbutton,Jtext field,Jtext area,Jradio button

**CHAPTER SIX: OVERVIEW OF CODES AND RESULT**:

**PROJECT CODE:**

**Com.failuredetection.controller**

Training controller.java

package com.failuredetection.controller;

import java.util.List;

import org.abego.treelayout.TreeForTreeLayout;

import org.abego.treelayout.util.DefaultTreeForTreeLayout;

import com.failuredetection.model.Node;

import com.failuredetection.model.TreeNode;

import com.failuredetection.treelayout.TextInBox;

import com.failuredetection.util.Constants;

import com.failuredetection.view.Main;

public class TrainingController {

public static TreeForTreeLayout<TextInBox> tree;

public TreeNode<Node> buildTree(final List<String> lines, Boolean isDetectionMode) {

TreeNode<Node> root = null;

Integer currentLevel = 0, index = 0;

TreeNode<Node> currentNode = null;

TextInBox rootBox = null;

TextInBox currentBox = null;

DefaultTreeForTreeLayout<TextInBox> tree = null;

try {

for (String line : lines) {

String[] splittedStr = line.split(" ");

// rootnode

if (index == 0) {

if (Constants.COMMAND\_CALL.equals(splittedStr[1])) {

Node node = new Node();

node.setLevel(++currentLevel);

node.setStartTime(Integer.parseInt(splittedStr[0]));

node.setName(splittedStr[2]);

root = new TreeNode<Node>(node);

rootBox = new TextInBox(node.getName());

currentNode = root;

currentBox = rootBox;

tree = new DefaultTreeForTreeLayout<TextInBox>(rootBox);

} else {

// TODO show error

}

} else {

if (Constants.COMMAND\_CALL.equals(splittedStr[1])) {

Node node = new Node();

node.setLevel(++currentLevel);

node.setStartTime(Integer.parseInt(splittedStr[0]));

node.setName(splittedStr[2]);

TreeNode<Node> treeNode = new TreeNode<Node>(node);

TextInBox tempTextInBox = new TextInBox(node.getName());

tree.addChild(currentBox, tempTextInBox);

currentBox = tempTextInBox;

currentNode.addChild(treeNode);

currentNode = treeNode;

} else {

Integer endTime = Integer.parseInt(splittedStr[0]);

Node node = currentNode.getData();

node.setEndTime(endTime);

node.setExecutionTime(endTime - node.getStartTime());

if(Boolean.TRUE.equals(isDetectionMode)) {

// Check if its deviating/abnormal or not

if((new Double(node.getExecutionTime())) < (Main.mean - 2 \* Main.standardDeviation) ||

(new Double(node.getExecutionTime())) > (Main.mean + 2 \* Main.standardDeviation)) {

return null;

}

}

currentNode = currentNode.getParent();

currentBox = tree.getParent(currentBox);

--currentLevel;

}

}

index++;

}

} catch (Exception e) {

TrainingController.tree = null;

return null;

}

TrainingController.tree = tree;

return root;

}

public Boolean matchTree(List<String> lines1, List<String> lines2) {

if(lines1.size() != lines2.size()) {

return Boolean.FALSE;

}

for(int i = 0; i < lines1.size(); i++) {

if(!(lines1.get(i).equals(lines2.get(i)))) {

return Boolean.FALSE;

}

}

return Boolean.TRUE;

}

}

C**om.Failuredetection.model**

**Node.java**

package com.failuredetection.model;

import java.util.ArrayList;

import java.util.List;

public class Node {

private String name;

private Integer executionTime;

private Integer startTime;

private Integer endTime;

private Integer level;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Integer getExecutionTime() {

return executionTime;

}

public void setExecutionTime(Integer executionTime) {

this.executionTime = executionTime;

}

public Integer getStartTime() {

return startTime;

}

public void setStartTime(Integer startTime) {

this.startTime = startTime;

}

public Integer getEndTime() {

return endTime;

}

public void setEndTime(Integer endTime) {

this.endTime = endTime;

}

public Integer getLevel() {

return level;

}

public void setLevel(Integer level) {

this.level = level;

}

public Node getClone() {

Node node = new Node();

node.setEndTime(this.endTime);

node.setExecutionTime(this.executionTime);

node.setLevel(this.level);

node.setName(this.name);

node.setStartTime(this.startTime);

return node;

}

}

**TreeNode.java**

package com.failuredetection.model;

import java.util.ArrayList;

import java.util.List;

public class TreeNode<T> {

private T data;

private List<TreeNode<T>> children;

private TreeNode<T> parent;

public TreeNode() {

super();

children = new ArrayList<TreeNode<T>>();

}

public TreeNode(T data) {

this();

setData(data);

}

public TreeNode<T> getParent() {

return this.parent;

}

public List<TreeNode<T>> getChildren() {

return this.children;

}

public int getNumberOfChildren() {

return getChildren().size();

}

public boolean hasChildren() {

return (getNumberOfChildren() > 0);

}

public void setChildren(List<TreeNode<T>> children) {

for(TreeNode<T> child : children) {

child.parent = this;

}

this.children = children;

}

public void addChild(TreeNode<T> child) {

child.parent = this;

children.add(child);

}

public void addChildAt(int index, TreeNode<T> child) throws IndexOutOfBoundsException {

child.parent = this;

children.add(index, child);

}

public void removeChildren() {

this.children = new ArrayList<TreeNode<T>>();

}

public void removeChildAt(int index) throws IndexOutOfBoundsException {

children.remove(index);

}

public TreeNode<T> getChildAt(int index) throws IndexOutOfBoundsException {

return children.get(index);

}

public T getData() {

return this.data;

}

public void setData(T data) {

this.data = data;

}

public String toString() {

return getData().toString();

}

@Override

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if (obj == null) {

return false;

}

if (getClass() != obj.getClass()) {

return false;

}

TreeNode<?> other = (TreeNode<?>) obj;

if (data == null) {

if (other.data != null) {

return false;

}

} else if (!data.equals(other.data)) {

return false;

}

return true;

}

/\* (non-Javadoc)

\* @see java.lang.Object#hashCode()

\*/

@Override

public int hashCode() {

final int prime = 31;

int result = 1;

result = prime \* result + ((data == null) ? 0 : data.hashCode());

return result;

}

}

# Com.Failuredetection.TreeLayout

**TextinBox.java**

package com.failuredetection.treelayout;

/\*\*

\* Represents a text to be displayed in a box of a given size.

\*

\* @author Udo Borkowski (ub@abego.org)

\*/

public class TextInBox {

public String text;

public final int height = 30;

public final int width = 60;

public TextInBox() {

}

public TextInBox(String text) {

this.text = text;

}

public TextInBox(String text, int width, int height) {

this.text = text;

// this.width = width;

// this.height = height;

}

}

**TextinboxNodeExtentProvider.java**

package com.failuredetection.treelayout;

import org.abego.treelayout.NodeExtentProvider;

/\*\*

\* A {@link NodeExtentProvider} for nodes of type {@link TextInBox}.

\* <p>

\* As one would expect this NodeExtentProvider returns the width and height as

\* specified with each TextInBox.

\*

\* @author Udo Borkowski (ub@abego.org)

\*/

public class TextInBoxNodeExtentProvider implements

NodeExtentProvider<TextInBox> {

@Override

public double getWidth(TextInBox treeNode) {

return treeNode.width;

}

@Override

public double getHeight(TextInBox treeNode) {

return treeNode.height;

}

}

**Com.FailureDetection.TreeLayout.view**

**SwingDemo.java**

public class SwingDemo {

private static void showInDialog(JComponent panel) {

JDialog dialog = new JDialog();

Container contentPane = dialog.getContentPane();

((JComponent) contentPane).setBorder(BorderFactory.createEmptyBorder(

10, 10, 10, 10));

contentPane.add(panel);

dialog.pack();

dialog.setLocationRelativeTo(null);

dialog.setVisible(true);

}

/\*\*

\* Shows a dialog with a tree in a layout created by {@link TreeLayout},

\* using the Swing component {@link TextInBoxTreePane}.

\*

\* @param args args[0]: treeName (default="")

\*/

public static void buildTree() {

// get the sample tree

TreeForTreeLayout<TextInBox> tree = TrainingController.tree;

// setup the tree layout configuration

double gapBetweenLevels = 50;

double gapBetweenNodes = 10;

DefaultConfiguration<TextInBox> configuration = new DefaultConfiguration<TextInBox>(

gapBetweenLevels, gapBetweenNodes);

// create the NodeExtentProvider for TextInBox nodes

TextInBoxNodeExtentProvider nodeExtentProvider = new TextInBoxNodeExtentProvider();

// create the layout

TreeLayout<TextInBox> treeLayout = new TreeLayout<TextInBox>(tree,

nodeExtentProvider, configuration);

// Create a panel that draws the nodes and edges and show the panel

TextInBoxTreePane panel = new TextInBoxTreePane(treeLayout);

showInDialog(panel);

}

}

**TextinBoxTreePane.java**

package com.failuredetection.treelayout.view;

import java.awt.Color;

import java.awt.Dimension;

import java.awt.FontMetrics;

import java.awt.Graphics;

import java.awt.geom.Rectangle2D;

import javax.swing.JComponent;

import org.abego.treelayout.TreeForTreeLayout;

import org.abego.treelayout.TreeLayout;

import com.failuredetection.treelayout.TextInBox;

/\*\*

\* A JComponent displaying a tree of TextInBoxes, given by a {@link TreeLayout}.

\*

\* @author Udo Borkowski (ub@abego.org)

\*/

public class TextInBoxTreePane extends JComponent {

private final TreeLayout<TextInBox> treeLayout;

private TreeForTreeLayout<TextInBox> getTree() {

return treeLayout.getTree();

}

private Iterable<TextInBox> getChildren(TextInBox parent) {

return getTree().getChildren(parent);

}

private Rectangle2D.Double getBoundsOfNode(TextInBox node) {

return treeLayout.getNodeBounds().get(node);

}

/\*\*

\* Specifies the tree to be displayed by passing in a {@link TreeLayout} for

\* that tree.

\*

\* @param treeLayout the {@link TreeLayout} to be displayed

\*/

public TextInBoxTreePane(TreeLayout<TextInBox> treeLayout) {

this.treeLayout = treeLayout;

Dimension size = treeLayout.getBounds().getBounds().getSize();

setPreferredSize(size);

}

// -------------------------------------------------------------------

// painting

private final static int ARC\_SIZE = 10;

private final static Color BOX\_COLOR = Color.orange;

private final static Color BORDER\_COLOR = Color.darkGray;

private final static Color TEXT\_COLOR = Color.black;

private void paintEdges(Graphics g, TextInBox parent) {

if (!getTree().isLeaf(parent)) {

Rectangle2D.Double b1 = getBoundsOfNode(parent);

double x1 = b1.getCenterX();

double y1 = b1.getCenterY();

for (TextInBox child : getChildren(parent)) {

Rectangle2D.Double b2 = getBoundsOfNode(child);

g.drawLine((int) x1, (int) y1, (int) b2.getCenterX(),

(int) b2.getCenterY());

paintEdges(g, child);

}

}

}

private void paintBox(Graphics g, TextInBox textInBox) {

// draw the box in the background

g.setColor(BOX\_COLOR);

Rectangle2D.Double box = getBoundsOfNode(textInBox);

g.fillRoundRect((int) box.x, (int) box.y, (int) box.width - 1,

(int) box.height - 1, ARC\_SIZE, ARC\_SIZE);

g.setColor(BORDER\_COLOR);

g.drawRoundRect((int) box.x, (int) box.y, (int) box.width - 1,

(int) box.height - 1, ARC\_SIZE, ARC\_SIZE);

// draw the text on top of the box (possibly multiple lines)

g.setColor(TEXT\_COLOR);

String[] lines = textInBox.text.split("\n");

FontMetrics m = getFontMetrics(getFont());

int x = (int) box.x + ARC\_SIZE / 2;

int y = (int) box.y + m.getAscent() + m.getLeading() + 1;

for (int i = 0; i < lines.length; i++) {

g.drawString(lines[i], x, y);

y += m.getHeight();

}

}

@Override

public void paint(Graphics g) {

super.paint(g);

paintEdges(g, getTree().getRoot());

// paint the boxes

for (TextInBox textInBox : treeLayout.getNodeBounds().keySet()) {

paintBox(g, textInBox);

}

}

**Com.DetectionFailure.util**

**Constants.java**

package com.failuredetection.util;

public interface Constants {

String COMMAND\_CALL = "C";

String COMMAND\_RETURN = "R";

}

**FileUtil.java**

package com.failuredetection.util;

import java.io.BufferedReader;

import java.io.File;

import java.io.FileReader;

import java.io.IOException;

import java.util.ArrayList;

import java.util.List;

public class FileUtil {

public static List<String> readFile(File file) {

List<String> lines = new ArrayList<String>();

BufferedReader br = null;

try {

br = new BufferedReader(new FileReader(file));

String line;

while ((line = br.readLine()) != null) {

lines.add(line);

}

} catch (Exception e) {

System.out.println("Unable to read file");

} finally {

if(br != null) {

try {

br.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

return lines;

}

}

**MathUtil.java**

package com.failuredetection.util;

import java.util.LinkedList;

import java.util.Queue;

import com.failuredetection.model.Node;

import com.failuredetection.model.TreeNode;

public class MathUtil {

public static Double calculateMean(TreeNode<Node> root) {

Double mean = 0.0;

int count = 0;

if (root != null) {

Queue<TreeNode<Node>> queue = new LinkedList<>();

queue.add(root); // push operation

++count;

while (!queue.isEmpty()) {

TreeNode<Node> treeNode = queue.poll();

queue.poll();

++count;

mean += treeNode.getData().getExecutionTime();

for (TreeNode<Node> child : treeNode.getChildren()) {

queue.add(child);

}

}

mean /= count;

}

return mean;

}

public static Double calculateStandardDeviation(TreeNode<Node> root, Double mean) {

Double standardDeviation = 0.0;

int count = 0;

if (root != null) {

Queue<TreeNode<Node>> queue = new LinkedList<>();

queue.add(root); // push operation

++count;

while (!queue.isEmpty()) {

TreeNode<Node> treeNode = queue.poll();

queue.poll();

++count;

standardDeviation += (new Double(treeNode.getData().getExecutionTime()) - mean)

\* (new Double(treeNode.getData().getExecutionTime()) - mean);

for (TreeNode<Node> child : treeNode.getChildren()) {

queue.add(child);

}

}

standardDeviation /= count;

}

return Math.sqrt(standardDeviation);

}

}

# Com.failuredetection.view

Main.java

package com.failuredetection.view;

import java.io.File;

import java.util.ArrayList;

import java.util.List;

import com.failuredetection.controller.TrainingController;

import com.failuredetection.model.Node;

import com.failuredetection.model.TreeNode;

import com.failuredetection.treelayout.view.SwingDemo;

import com.failuredetection.util.FileUtil;

import com.failuredetection.util.MathUtil;

import javafx.application.Application;

import javafx.event.ActionEvent;

import javafx.event.EventHandler;

import javafx.geometry.Pos;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.layout.VBox;

import javafx.stage.FileChooser;

import javafx.stage.Stage;

public class Main extends Application {

TrainingController trainingController = new TrainingController();

TreeNode<Node> root = null;

public static List<String> lines = new ArrayList<String>();

public static Double mean = 0.0, standardDeviation = 0.0;

@Override

public void start(Stage primaryStage) throws Exception {

Button trainingBtn = new Button("Training Mode");

trainingBtn.setMinHeight(100);

trainingBtn.setMinWidth(150);

trainingBtn.setMaxHeight(100);

trainingBtn.setMaxWidth(150);

trainingBtn.setPrefHeight(100);

trainingBtn.setPrefWidth(150);

trainingBtn.setStyle("-fx-color: gray;");

Button detectionBtn = new Button("Detection Mode");

detectionBtn.setMinHeight(100);

detectionBtn.setMinWidth(150);

detectionBtn.setMaxHeight(100);

detectionBtn.setMaxWidth(150);

detectionBtn.setPrefHeight(100);

detectionBtn.setPrefWidth(150);

detectionBtn.setStyle("-fx-color: gray;");

final FileChooser fileChooser = new FileChooser();

trainingBtn.setOnAction(new EventHandler<ActionEvent>() {

@Override

public void handle(ActionEvent event) {

// Open file chooser

File file = fileChooser.showOpenDialog(primaryStage);

if (file != null) {

lines = FileUtil.readFile(file);

root = trainingController.buildTree(lines, Boolean.FALSE);

if(TrainingController.tree != null) {

SwingDemo.buildTree();

}

Main.mean = MathUtil.calculateMean(root);

Main.standardDeviation = MathUtil.calculateStandardDeviation(root, mean);

}

System.out.println("Done");

}

});

detectionBtn.setOnAction(new EventHandler<ActionEvent>() {

@Override

public void handle(ActionEvent event) {

// Open file chooser

File file = fileChooser.showOpenDialog(primaryStage);

if (file != null) {

List<String> lines1 = FileUtil.readFile(file);

// Match lines

Boolean matched = trainingController.matchTree(lines, lines1);

// Check Mean and standard deviation

TreeNode<Node> root1 = trainingController.buildTree(lines1, Boolean.TRUE);

if(Boolean.TRUE.equals(matched) && root1 != null) {

// Show Pass

Button passBtn = new Button("Pass");

passBtn.setMinHeight(100);

passBtn.setMinWidth(150);

passBtn.setMaxHeight(100);

passBtn.setMaxWidth(150);

passBtn.setPrefHeight(100);

passBtn.setPrefWidth(150);

passBtn.setStyle("-fx-color: green;");

VBox vBox = new VBox(50);

vBox.setAlignment(Pos.CENTER);

vBox.getChildren().addAll(passBtn);

Scene scene = new Scene(vBox);

Stage stage = new Stage();

primaryStage.close();

stage.setTitle("Result");

stage.setScene(scene);

stage.setWidth(500);

stage.setHeight(500);

stage.show();

} else {

// Show Fail

Button failBtn = new Button("Fail");

failBtn.setMinHeight(100);

failBtn.setMinWidth(150);

failBtn.setMaxHeight(100);

failBtn.setMaxWidth(150);

failBtn.setPrefHeight(100);

failBtn.setPrefWidth(150);

failBtn.setStyle("-fx-color: red;");

VBox vBox = new VBox(50);

vBox.setAlignment(Pos.CENTER);

vBox.getChildren().addAll(failBtn);

Scene scene = new Scene(vBox);

Stage stage = new Stage();

primaryStage.close();

stage.setTitle("Result");

stage.setScene(scene);

stage.setWidth(500);

stage.setHeight(500);

stage.show();

}

}

}

});

VBox vBox = new VBox(50);

vBox.setAlignment(Pos.CENTER);

vBox.getChildren().addAll(trainingBtn, detectionBtn);

Scene scene = new Scene(vBox);

primaryStage.setTitle("Mode");

primaryStage.setScene(scene);

primaryStage.setWidth(500);

primaryStage.setHeight(500);

primaryStage.show();

}

public static void main(String[] args) {

launch(args);

}

}

**Result.java**

package com.failuredetection.view;

public class Result {

}

**Input.txt:**

The input of the system is a trace file.Each line indicates function call or return .The format of trace file is

<timestamp> <C or r> <Target address if it is a function call>

1000 C funcA

1010 C funcB

1020 C funcC

1030 R

1040 C funcD

1050 R

1060 C funcE

1070 C funcF

1080 R

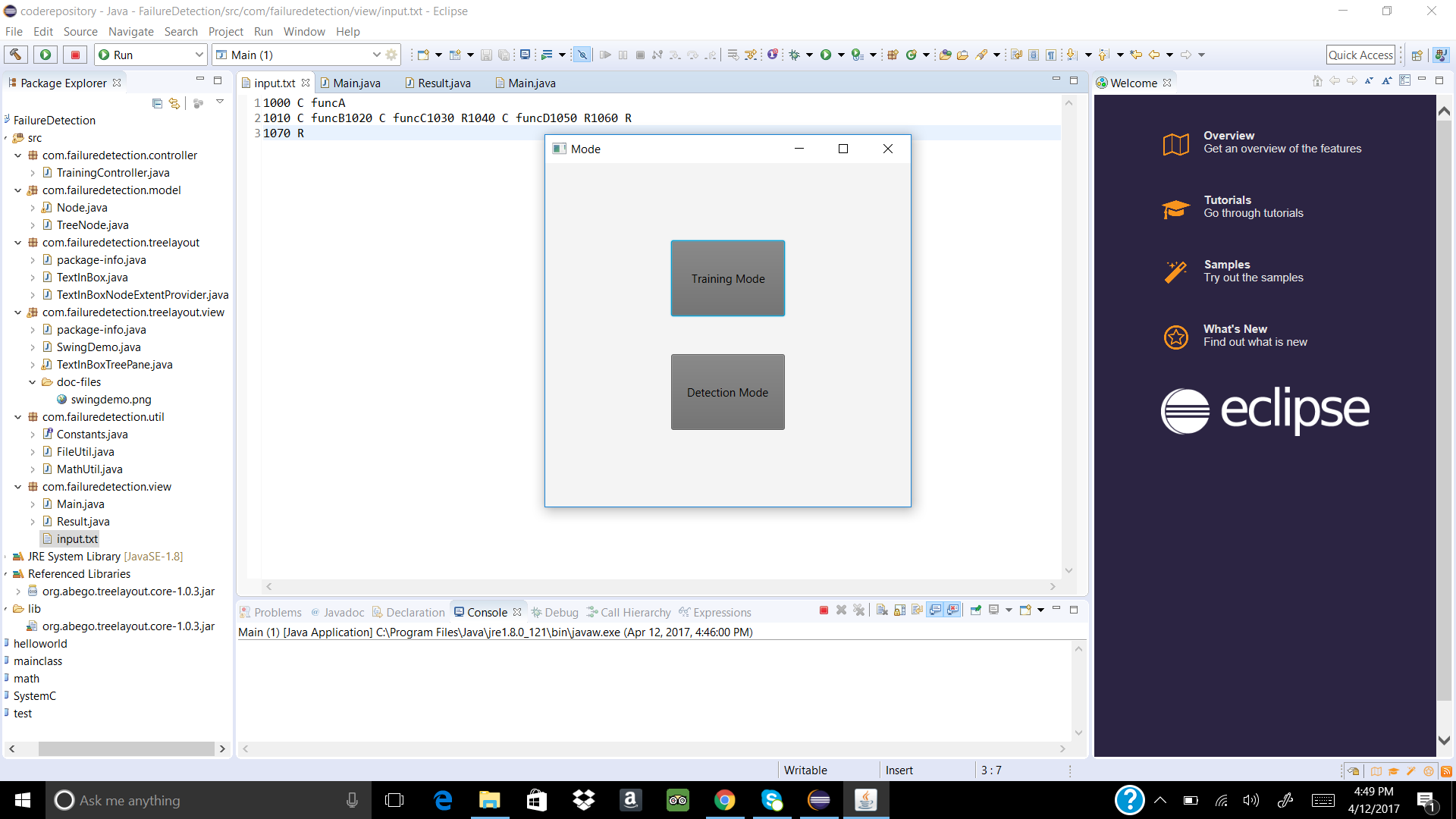
1090 R

10100 R

10110 R

**RESULTS:**

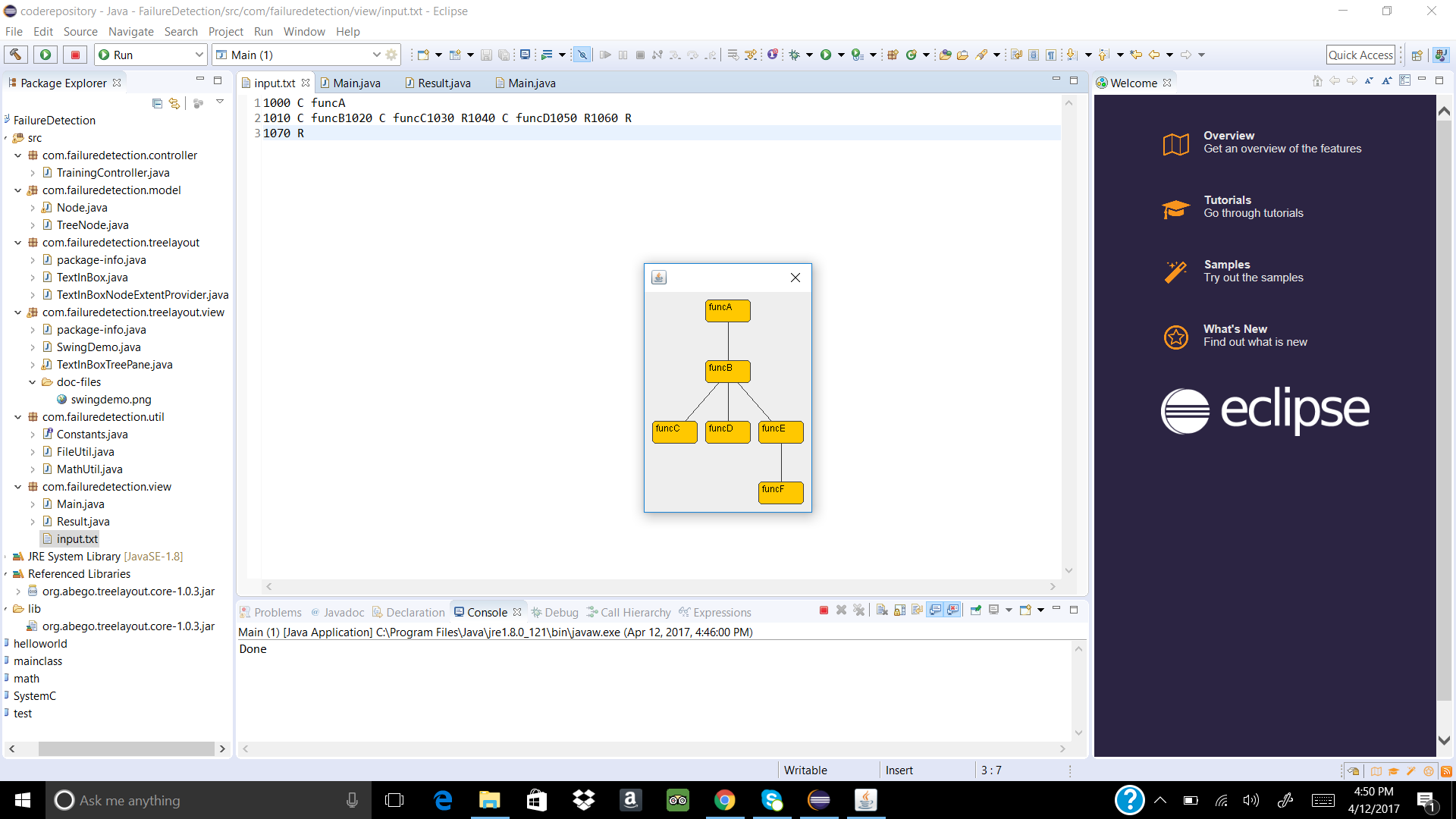
**Output screenshots in java eclipse**

****

**Fig 6****:** GUI Displayed with training and detection mode

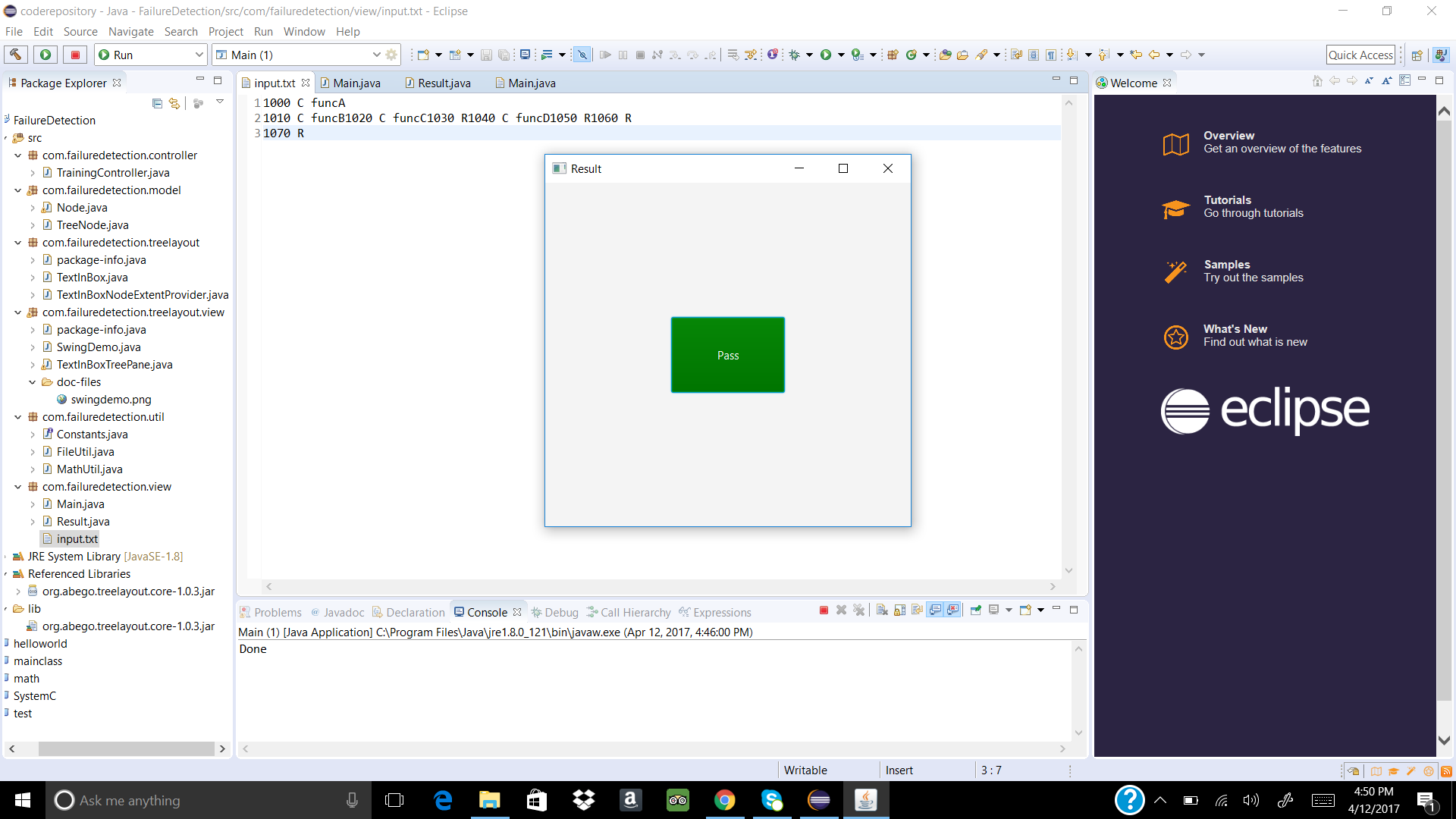
**Training mode:**

When the input trace file is given in training mode it constructs a call stack tree as displayed

****

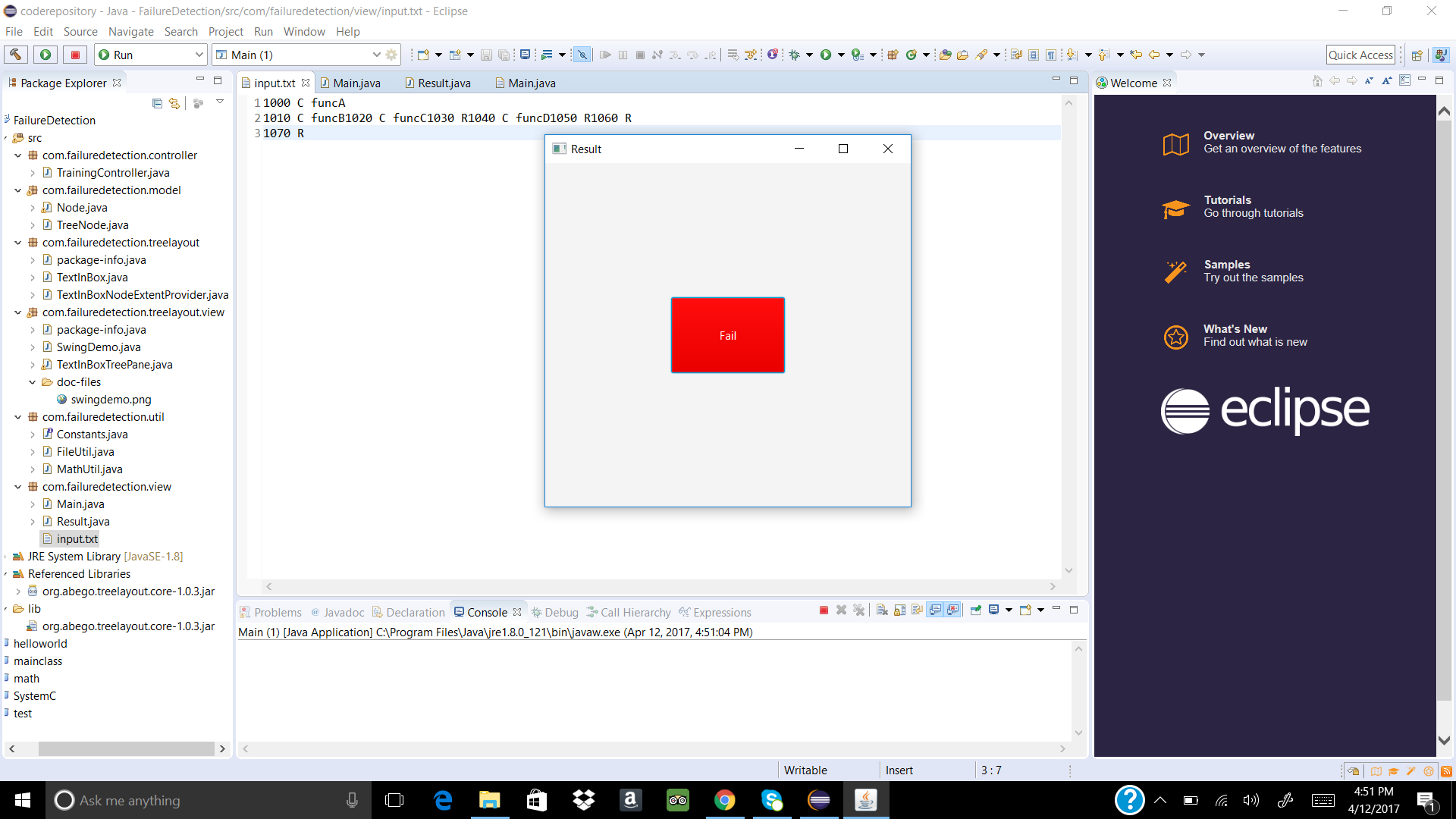
**Fig 7:** call stack tree is displayed

# Detection output: If the built call stack tree matchs with online call stack it gives “pass” or “fail”

**Passcase:**

**Fig 8:** output for pass

**Fail case:**

****

**Fig 9:** output for fail

# CONCLUSION: A GUI-based application will be developed .The call stack tree is built when the Iot device is being tested .The time when the call stack tree is being built as training mode. Comparing the call stack tree with the online call stack collected by call stack monitor as detection mode. Thus if it does not match a fault is declared

# REFERENCES :

[1] https://docs.oracle.com/javase/tutorial/uiswing/components/tabbedpane.html

[2] https://docs.oracle.com/javase/7/docs/api/javax/swing/JTabbedPane.html

[3] https://docs.oracle.com/javase/7/docs/api/java/awt/Graphics.html

[4] http://docstore.mik.ua/orelly/java-ent/jfc/ch09\_01.htm

[5] https://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html

[6] https://docs.oracle.com/javase/tutorial/uiswing/

[7] https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html

https://en.wikipedia.org/wiki/Call\_stack

[8] https://docs.oracle.com/javase/7/docs/api/javax/swing/JMenu.html

[9] http://www.wideskills.com/java-tutorial/java-button-class-example

[10] https://docs.oracle.com/javase/7/docs/api/javax/swing/JPanel.html

# VITA

Sai Chandraneel Dodda is an international student from Hyderabad, India. He completed his Bachelor’s degree in Electronics and Communication Engineering at GITAM University, India. He is presently pursuing his Master’s degree in Electrical Engineering at The University of Texas at San Antonio.