IDENTIFYING FAULT LOCATION

By

Rahul Batheja (051319)

Gurdanish Bawa (051324)

विद्या तत्त्व ज्योतिसमः
JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY

MAY-2009

Submitted in partial fulfillment of the Degree of Bachelor of Technology

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING & INFORMATION TECHNOLOGY

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY-WAKNAGHAT

CERTIFICATE

We hereby certify that the work which is being presented in the project report entitled **"Identifying Fault Location"** by **"Rahul Batheja and Gurdanish Bawa"** in partial fulfillment of requirements for the award of degree of B. Tech. (C.S.E.) submitted in the Department of (Computer Science Engineering & Information Technology) at JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY is an authentic record of our work carried out during a period from 04-08-2008 to 27-05-2009 under the supervision of **Ms. Maneesha Srivastav**. The matter presented in this project report has not been submitted by us in any other University / Institute for the award of B. Tech. Degree.

[illegible]

Rahul Batheja (051319)

[illegible]

Gurdanish Bawa (051324)

This is to certify that the above statement made by the candidates is correct to the best of my knowledge.

[illegible]

Ms. Maneesha Srivastav

ACKNOWLEDGMENT

We feel great pleasure in expressing our regards and gratitude to Ms. Maneesha Srivastav, Lecturer Computer Science Department, Jaypee University of Information Technology for giving us an opportunity to prepare a project for our final year.

We are indebted to her for bequeathing us her vast knowledge and generous guidance at every stage of our project. Her advice, suggestions and encouragement led us through this project presentation.

[illegible]

Rahul Batheja

[illegible]

Gurdanish Bawa

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

The debugging tool helps the user to find the location of fault in the program. The Tarantula technique is used to develop this tool which is based on calculation of suspiciousness value for each executable statement in the code. Suspiciousness for any statement is the value which signifies danger because if the statements are passed primarily by failed test cases, the statement is highly suspicious of being faulty. The code entered by the user is edited to indentify which line is executed at the time of compilation. The tool stores the record of each line that is executed by the given test cases and based on the failed or passed test case entered by the user, the suspiciousness value is calculated, according to which ranks are assigned to each executable line. The line having higher rank has the maximum probability of having a fault.

CHAPTER-1

INTRODUCTION

1.1 Debugging

A practicing programmer inevitably spends a lot of time tracking down and fixing bugs. Debugging, particularly debugging of other people's code, is a skill separate from the ability to write programs in the first place. Unfortunately, while debugging is often practiced, it is rarely taught. A typical course in debugging techniques consists merely of reading the manual for a debugger.

Debugging is a methodical process of finding and reducing the number of bugs, or defects, in a computer program or a piece of electronic hardware thus making it behave as expected. Debugging tends to be harder when various subsystems are tightly coupled, as changes in one may cause bugs to emerge in another.

In computers, debugging is the process of locating and fixing or bypassing bugs (errors) in computer program code or the engineering of a hardware device. To debug a program or hardware device is to start with a problem, isolate the source of the problem, and then fix it. A user of a program that does not know how to fix the problem may learn enough about the problem to be able to avoid it until it is permanently fixed. When someone says they've debugged a program or "worked the bugs out" of a program, they imply that they fixed it so that the bugs no longer exist.

Debugging is a necessary process in almost any new software or hardware development process, whether a commercial product or an enterprise or personal application program. For complex products, debugging is done as the result of the unit test for the smallest unit of a system, again at component test when parts are brought together, again at system test when the product is used with other existing products, and again during customer beta test, when users try the product out in a real world situation. Because most computer programs and many programmed hardware devices contain thousands of lines of code, almost any new product is likely to contain a few bugs. Invariably, the bugs in the functions that get most use are found and fixed first. An early version of a program that has lots of bugs is referred to as "buggy."

Debugging tools (called debuggers) help identify coding errors at various development stages. Some programming language packages include a facility for checking the code for errors as it is being written.

1.2 Debugging tool

A tool used by programmers to reproduce failures, investigate the state of programs and find the corresponding defect. Debuggers enable programmers to execute programs step by step, to halt a program at any program statement and to set and examine program variables.

1.3 Debugger

A debugger is a computer program that is used to test and debug other programs. The code to be examined might alternatively be running on an instruction set simulator (ISS), a technique that allows great power in its ability to halt when specific conditions are encountered but which will typically be much slower than executing the code directly on the appropriate processor.

In other words debugger can be defined as “A special program used to find errors (bugs) in other programs. A debugger allows a programmer to stop a program at any point and examine and change the values of variables.”

When the program crashes, the debugger shows the position in the original code if it is a source-level debugger or symbolic debugger, commonly seen in integrated development environments. If it is a low-level debugger or a machine-language debugger it shows the line in the disassembly. (A "crash" happens when the program cannot continue because of a programming bug. For example, perhaps the program tried to use an instruction not available on the current version of the CPU or attempted access to unavailable or protected memory.)

Typically, debuggers also offer more sophisticated functions such as running a program step by step (single-stepping), stopping (breaking) (pausing the program to examine the current state) at some kind of event by means of breakpoint, and tracking the values of some variables. Some debuggers have the ability to modify the state of the program while it is running, rather than merely to observe it.

The importance of a good debugger cannot be overstated. Indeed, the existence and quality of such a tool for a given language and platform can often be the deciding factor in its use, even if another language/platform is better-suited to the task. However, it is also important to note that software can (and often does) behave differently running under a debugger than normally, due to the inevitable changes the presence of a debugger will make to a software program's internal timing. As a result, even with a good debugging tool, it is often very difficult to track down runtime problems in complex multi-threaded or distributed systems.

The same functionality which makes a debugger useful for eliminating bugs allows it to be used as a software cracking tool to evade copy protection, digital rights management, and other software protection features.

Most mainstream debugging engines, such as gdb and dbx provide console-based command line interfaces. Debugger front-ends are popular extensions to debugger engines that provide IDE integration, animation, and visualization features.

1.3.A *List of debuggers*

AppPuncher Debugger: It is used to debug Rich Internet Application. RIA AppPuncher™ is a software testing and debugging product. AppPuncher was designed from the ground up to facilitate testing of the Rich Internet Applications. The product provides the functionality such as AppPuncher supports Flash, Flex, Silverlight, AJAX clients as well as traditional browser-based applications. From the client/server protocol perspective, the product enables testing of the client/server applications using any of the following protocols: HTTP, SOAP, REST/XML, AMF and RTMP. AppPuncher runs on Windows, MAC OS X and *nix platforms.

CA/EZTEST (Cics Interactive test/debug): CA/EZTEST was a CICS interactive test/debug software package distributed by Computer Associates and originally called EZTEST/CICS, produced by Capex Corporation of Phoenix, Arizona with assistance from Ken Dakin from England. The product provided Source level test and debugging features for programs written in COBOL, PL/1 and Assembler languages to complement their own existing COBOL optimizer product.

CodeView: CodeView was a standalone debugger created by David Norris at Microsoft in 1985 as part of its development toolset. It originally shipped with Microsoft C 4.0 and later. It also shipped with Visual Basic for MS-DOS, Microsoft Basic PDS, and a number of other Microsoft language products. It was one of the first debuggers on the MS-DOS platform that was full-screen oriented, rather than line oriented.

DBG (a PHP Debugger and Profiler): Works transparently, neither script nor PHP engine modifications required. server part (dbg module) runs on all platforms where PHP itself runs. It works transparently across the global network as well as locally. JIT, when enabled it can start debugging Just In Time when an error happens. It also supports back-trace, e.g. displays a list of all procedures with their local variables, the current execution position reached from. In other words you can watch local variables or function parameters in all active and nested scopes. Certainly, you can execute script in the debugger step by step (step-in, step-out, step-over, run to cursor, change execution point withing current scope...), evaluate any valid php expression(s) or inspect arrays, classes and simple variables, modify their values on the fly and even create any new variables. Dbg supports conditional breakpoints and even global ones (commercial version only). Breakpoints can be skipped specified number of times.

DBX: DBX is a popular Unix-based source-level debugger found primarily on Solaris, AIX, IRIX, and BSD Unix systems. It provides symbolic debugging for programs written in C, C++, Pascal, and Fortran. Useful features include stepping through programs one source line or machine instruction at a time. In addition to simply viewing operation of the program, variables can be manipulated and a wide range of expressions can be evaluated and displayed.

DDD (Data Display Debugger): Data Display Debugger, or DDD, is a popular free software (under the GNU GPL) graphical user interface for command-line debuggers such as GDB, DBX, JDB, WDB, XDB, the Perl debugger, the Bash debugger, the Python debugger, and the GNU Make debugger. DDD has GUI front-end features such as viewing source texts and its interactive graphical data display, where data structures are displayed as graphs.

Dynamic debugging technique (DDT): Dynamic Debugging Technique, or DDT, was the name of several debugger programs originally developed for

DEC hardware, initially known as DEC Debugging Tape because it was distributed on paper tape). The first version of DDT was developed at MIT for the PDP-1 computer in 1961, but newer versions on newer platforms continued to use the same name. After being ported to other vendor's platforms and changing media, the name was changed to the less DEC-centric version.

GNU Debugger (GDB): GDB, the GNU Project debugger, allows you to see what is going on `inside' another program while it executes -- or what another program was doing at the moment it crashed.

Nemiver — Nemiver is a graphical debugger for GNOME, based on gdb. It is intended to be a no-nonsense debugger, allowing you get things done without requiring you to remember any arcane command names or key combinations. It is written in C++, using the gtkmm toolkit.MacsBug

OLIVER: OLIVER (CICS interactive test/debug) was a proprietary testing and debugging toolkit for interactively testing programs designed to run on IBM's Customer Information Control System (CICS) on IBM's System architecture.

RealView Debugger: Commercial debugger produced for and designed by ARM. The debugger in the RealView Development Suite delivers outstanding visibility of the behavior of software and hardware within complex SoCs. As part of the RealView Development Suite, the debugger offers support for all ARM architectures, including the latest Cortex family of processors with CoreSight on-chip debug and trace technology.

Turbo Debugger: Turbo Debugger was a machine-level debugger for MS-DOS executables sold by Borland. This tool provided a full-screen debugger with powerful capabilities for watching the execution of instructions, monitoring machine registers, etc. Later versions are able to step through source code compiled with Borland compilers set to provide debugging information.

Zeta Debugger: Zeta Debugger is a stand-alone source level debugger and code profiler for Windows 98/2000/XP applications written in C/C++ or assembly languages. Source level debugging is allowed when symbolic debug information emitted by your compiler is one of those supported by our debugger or external plug-in modules. Otherwise, when this information is absent or not recognized, you can only debug at machine level.

1.4 Bug

Bug in computer science, an error in software or hardware. In software, a bug is an error in coding or logic that causes a program to malfunction or to produce incorrect results. Minor bugs—for example, a cursor that does not behave as expected—can be inconvenient or frustrating, but not damaging to information. More severe bugs can cause a program to "hang" (stop responding to commands) and might leave the user with no alternative but to restart the program, losing whatever previous work had not been saved. In either case, the programmer must find and correct the error by the process known as debugging. Because of the potential risk to important data, commercial application programs are tested and debugged as completely as possible before release. Minor bugs found after the program becomes available are corrected in the next update; more severe bugs can sometimes be fixed with special software, called patches, that circumvents the problem or otherwise alleviates its effects. In hardware, a bug is a recurring physical problem that prevents a system or set of components from working together properly. The origin of the term reputedly goes back to the early days of computing, when a hardware problem in an electromechanical computer at Harvard University was

(Entomologists will undoubtedly be quick to note that a moth is not really a bug.)

1.5 <u>Project Scope</u>

The project aims at developing a debugging tool which can be used to identify the location of fault in the program entered by the user using a debugging technique. The user will enter the file name (which is to be debugged) as input to the program, the file is then modified that will check which line of the code is reached by giving various test cases as an input to the program. The value of suspiciousness and rank is calculated corresponding to each line of code entered by the user. Higher the suspiciousness more is the probability of a bug being lying there and vice-versa.

CHAPTER-2

DEBUGGING PROCESS

2.1 Introduction

Often the first step in debugging is to attempt reproduce the problem. This can be a non-trivial task, for example in case of parallel processes or some unusual software bugs. Also specific user environment and usage history can make it difficult to reproduce the problem.

After the bug is reproduced, the input of the program needs to be simplified to make it easier to debug. For example, a bug in a compiler can make it crash when parsing some large source file. However, after simplification of the test case, only few lines from the original source file can be sufficient to reproduce the same crash. Such simplification can be made manually, using a divide-and-conquer approach. The programmer will try to remove some parts of original test case and check if the problem still exists. When debugging the problem in GUI, the programmer will try to skip some user interaction from the original problem description and check if remaining actions are sufficient for bug to appear. To automate test case simplification, delta debugging methods can be used.

After the test case is sufficiently simplified, a programmer can use a debugger to examine program states (values of variables, the call stack) and track down the origin of the problem. Alternatively tracing can be used. In simple case, tracing is just a few print statements, which output the values of variables in certain points of program execution.

Remote debugging is the process of debugging a program running on a system different than the debugger. To start remote debugging, debugger connects to a remote system over a network. Once connected, debugger can control the execution of the program on the remote system and retrieve information about its state.

Post-mortem debugging is the act of debugging the core dump of process. The dump of the process space may be obtained automatically by the system, or manually by the interactive user. Crash dumps (core dumps) are often generated after a process has terminated due to an unhandled exception.

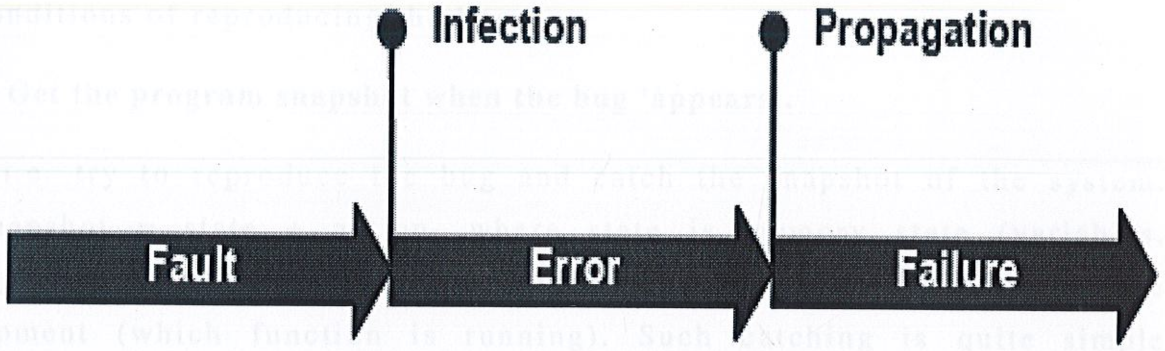


Figure 1: The relation between fault error and failure

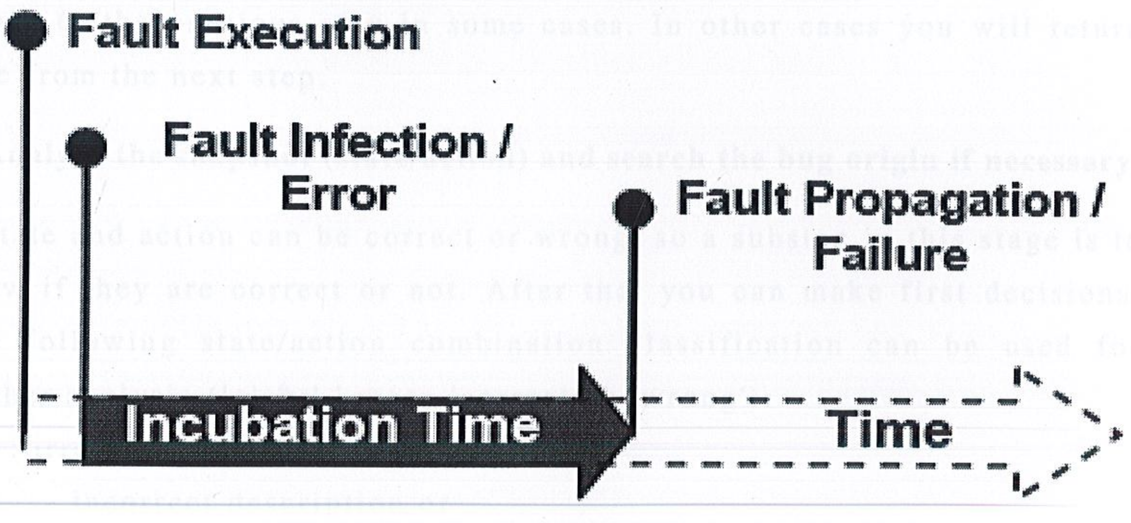


Figure 2: The incubation time of a fault

2.2 The Process

Unlike development debugging is iterative in itself (It means development in general, but in most more or less complex projects development can and even should be iterative too). Let's describe its three main stages:

1. Get the bug description.

Maybe this stage doesn't belong to debugging itself, but it's so important that we have mentioned it here too. You should get as more details from the user as possible: product version, OS version, all conditions of reproducing the bug.

2. Get the program snapshot when the bug 'appears'.

i.e. try to reproduce the bug and catch the snapshot of the system. Snapshot = state + action, where state is memory state (variables, registers, files, etc.) and action is what the program is doing at the moment (which function is running). Such catching is quite simple sometimes, e.g. the program crashes and you can just switch to your debugger to see what the problem is. But in some cases you have to perform many iterations to catch the snapshot. You maybe even have to return to the previous step in some cases. In other cases you will return here from the next step.

3. Analyze the snapshot (state/action) and search the bug origin if necessary.

State and action can be correct or wrong, so a substep in this stage is to know if they are correct or not. After that you can make first decisions. The following state/action combination classification can be used for further analysis ('+' & '-' mean 'correct' & 'wrong'):

+/+ : irreproducible bug; you have:

- incorrect description or
- different environments (OS, etc.)

+/- : a simply found bug, you have to:

- add some checking (NULL pointer, divide by zero, etc.) or
- additional implementation of something

-/+ : a bug requiring searching of its source, origin;

- methods you can use:
- tracing from previous moments
- tracing of places where the state components are used/changed
- tracing with different input data
- and your experience is desired:
- key places in the program (e.g., the start of user input processing)

-/- : same as -/+ plus additional difficulty, but due to the difficulty of situation in some cases some work-around may be enough decision, i.e. making action correct (+).

4. Fix the bug.

Ideas about fixing may appear on any stage of debugging process, but only that one may be implemented, which is proved, i.e. it:

1) fixes the class of bugs which the described one belongs to, and
2) doesn't introduce other bugs: in other places and/or with other input data.

Experience i.e. knowledge of the program plays the key role in all stages of debugging. Some of

the most important components of it is knowledge of key places of the program, this knowledge can help both in getting program snapshot and searching the bug origin. Examples of key places: the start of user input processing, multi-functional procedures and so on. Therefore the following ideas can be useful.

2.3 Identifying The Bug:

For a debugger to work efficiently it is very important to identify the bug in the source code so that the bug can be removed. To identify the location of bug in the user input program, we have used the tarantula debugging technique. In modern software development, testing and debugging software is done as part of an integrated method of software development. An appropriate method of bug finding can easily help developers locate and remove bugs. A software bug is regarded as the abnormal program behaviors which deviates from its specification, including poor performance when a threshold level of performance is included as part of specification. Bug patterns, which are related to anti-patterns, are recurring relationships between potential bugs and explicit errors in a program; they are common coding practices which share the similar symptoms and have been proven to fail time and time again. Those bug patterns are raised from the misunderstanding of language features, the misuse of positive design patterns or simple mistakes having the common behaviors. Such bug patterns are an essential complement to the traditional design pattern, just as a good programmer needs to know design patterns which can be applied in various context and improve the software quality, also to be a good software developer or problem solver the knowledge of common causes of faults is a need in order to know how to fix the software bugs.

We may not know what types of bugs are unique without a proper bug pattern classification, and this poses several restrictions on the research and development of programs in the language:

• Developers do not know what kind of bugs are most likely to happen in a program, and therefore do not know how to prevent them. In other words, programmer would be lack of a fundamental knowledge on how to write bug-free code.

• Testers do not have sufficed knowledge of how to write adequacy test cases that can effectively cover most of the common potential errors. Only when having an idea of how the common bugs happened in programs, can tester set up criteria for better addressing the specific bugs?

• Software maintenance staffs do not know which features of the language are more likely to result in the faulty code; so they cannot have a clear view on the current system when doing the maintenance tasks.

CHAPTER 3

PROJECT OBJECTIVE

3.1 Objective

In the program development, it is unavoidable to localize the bugs. For an intuitive viewpoint, we call symptoms the appearance of an anomaly during the execution of a program. Symptoms are caused by errors in the program. An error is a piece of code. Strictly speaking, error localization, when a symptom is given, is error diagnosis and it can be seen as a first step in debugging, a second step being error correction.

The objective of our project is to design a debugging tool that act as an automated error detection help. It will help the user to identify the location of the bug in his/her source code. The idea is to provide programmers with a means of analyzing the execution the program and tracking problems in order to optimize execution time and utilization of resources. The user will enter the code which is to be debugged, and the tool helps the user to locate the bugs in the code by pointing out the exact position where there is a possibility of having an error.

3.2 Our Approach

Our approach to the development of debugging tool consists of following:

1. Analysis
2. Implementation
3. GUI

3.2.1 *Analysis*

During analysis phase we searched for various debugging techniques and short selected a few for implementation. Our project is based on implementation of debugging technique. We have studied various techniques of debugging and the following section describes some of the techniques that we had short listed and their drawbacks while implementing these techniques.

Debugging Techniques:

A) Parallel Debugging Technique

B) Holistic Debugging Technique

C) Run-time Debugging Technique

D) Tarantula Debugging Technique

3.2.1.A *Parallel Debugging Technique*

The presence of multiple faults in a program can inhibit the ability of fault-localization techniques to locate the faults. This problem occurs for two reasons: when a program fails, the number of faults is, in general, unknown; and certain faults may mask or obfuscate other faults. Debugging software is an expensive and mostly manual process. This debugging expense has two main dimensions: the labor cost to discover and correct the bugs, and the time required to produce a failure-free program.1 A developer generally wants to find a good trade-off between these dimensions that reflects the developer's resources and tolerance for delay. Of all debugging activities, fault localization is among the most expensive. Any improvement in the process of finding faults will generally decrease the expense of debugging. In practice, developers are aware of the number of failed test cases for their programs, but are unaware of whether a single fault or many faults caused those failures. Thus, developers usually target one fault at a time in their debugging. A developer can inspect a single failed test case to attempt to find its cause using an existing debugging technique, or she can utilize all failed test cases using a fault-localization technique. After a fault is found and fixed, the program must be retested to determine whether previously failing test cases now pass. If failures remain, the debugging process is repeated. We call this one-fault at-a-time mode of debugging and retesting sequential debugging.

It automatically partitions the set of failing test cases into clusters that target different faults, called fault-focusing clusters, using behavior models and fault-localization information created from execution data. Each fault-focusing cluster is then combined with the passing test cases to get a specialized test suite that targets a single fault. Consequently, specialized test suites based on fault-focusing clusters can be assigned to developers who can then debug multiple faults in parallel. The resulting specialized test suites provide a prediction of the number of current, active faults in the program. In practice, however, there may be more than one developer available to debug a program, particularly under urgent circumstances such as an imminent release date. Because, in general, there may be multiple faults whenever a program fails on a test suite, an effective way to handle this situation is to create parallel work flows so that multiple developers can each work to isolate different faults, and thus, reduce the overall time to a failure-free program. Like the parallelization of other work flows, such as computation, the principal problem of providing parallel work flows in debugging is determining the partitioning and assignment of subtasks. To perform the partitioning and assignment requires an automated technique that can detect the presence of multiple faults and map them to sets of failing test cases (i.e., clusters) that can be assigned to different developers.

It is used in those systems that have complex software or hardware architecture. It simultaneously debugs a program for multiple faults. The main benefit of parallel debugging technique is that it can result in decreased time to a failure-free program; our empirical evaluation supports this savings for our subject program. When resources are available to permit multiple developers to debug simultaneously, which is often the case, specialized test suites based on fault-focusing clusters can substantially reduce the time to a failure-free program while also reducing the number of testing iterations and their related expenses. Another benefit is that the fault-localization effort within each cluster is more efficient than without clustering. Thus, the debugging effort yields improved utilization of developer time, even if performed by a single developer. Our empirical evaluation shows that, for our subject, using the clusters provides savings in effort, even if debugging is done sequentially. A third benefit is that the number of clusters is an early estimate of the number of existing active faults. A final benefit is that our technique automates a debugging process that is already naturally occurs in current practice. For example, on bug-tracking systems for open-source projects, multiple developers are assigned to different faults, each working with a set of inputs that cause different known failures. The technique improves on this practice in a number of ways. First, the current practice requires a set of coordinating developers who triage failures to determine which appear to exhibit the same type of behavior. Often, this process involves the actual localization of the fault to determine the reason that a failure occurred, and thus a considerable amount of manual effort is needed. Our techniques can categorize failures automatically, without the intervention of the developers. This automation can save time and reduce the necessary labor involved. Second, in the current practice, coordinating developers categorize failures based on the failure output. Our techniques look instead at the execution behavior of the failures, such as how control flowed through the program, which may provide more detailed and rich information about the executions. Third, the current practice involves developers finding faults that cause failures using tedious, manual processes such as using print statements and symbolic debuggers on a single failed execution. Our techniques can automatically examine a set of failures and suggest likely fault locations in the program.

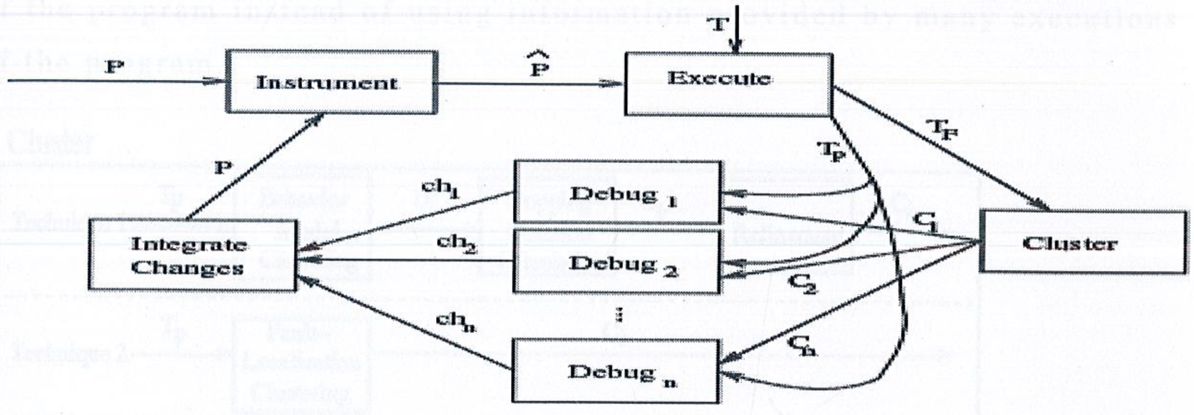
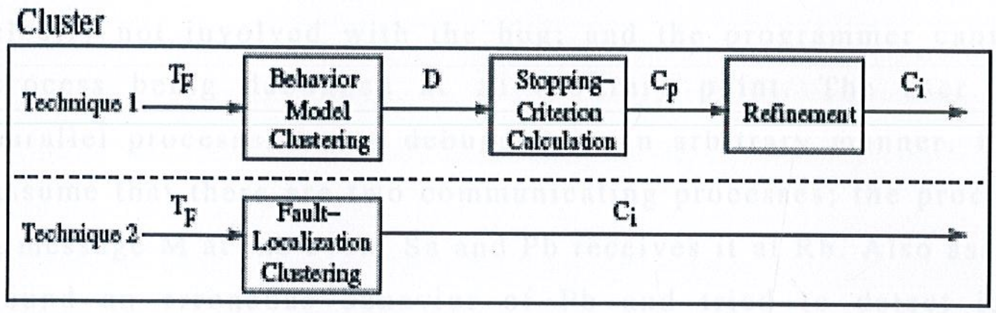


Figure 3: Technique for debugging in parallel.

To simultaneously debug multiple faults in parallel, we defined a parallel-debugging process, which is shown by the dataflow diagram in above figure. The program under test, P, is instrumented to produce P^. When P^ is executed with test suite T, it produces a set of passing test cases TP and a set of failing test cases TF, along with execution information; such as branch or method profiles. TF and the execution information are input to the clustering technique, Cluster, to produce a set of fault-focused clusters C1, C2, ...,Cn that are disjoint subsets of TF. Each Ci is combined with TP to produce a specialized test suite that assists in locating a particular fault. Using these test suites, developers can debug the program in parallel—shown as Debugging in the figure. The resulting changes, ch1, ch2, ..., chn, are integrated into the program. This process can be repeated until all test cases pass.

In practice, software developers locate faults in their programs using a highly involved, manual process. This process usually begins when the developers run the program with a test case (or test suite) and observe failures in the program. The developers then choose a particular failed test case to run, and iteratively place breakpoints using a symbolic debugger, observe the state until an erroneous state is reached, and backtrack until the faults are found. This process can be time-consuming and ad-hoc. Additionally, this process uses results of only one execution of the program instead of using information provided by many executions of the program.



Cluster
Technique 1
T_F
Behavior Model Clustering
D
Stopping-Criterion Calculation
C_p
Refinement
C_i
Technique 2
T_F
Fault-Localization Clustering
C_i

Figure 4: Two alternative techniques to cluster failed test cases for parallel debugging.

TP and a set of failing test cases TF , along with execution information, such as branch or method profiles. TF and the execution information are input to the clustering technique, Cluster, to produce a set of fault-focused clusters C1, C2, ..., Cn that are disjoint subsets of TF . Each Ci is combined with TP to produce a specialized test suite that assists in locating a particular fault. Using these test suites, developers can debug the program in parallel—shown as Debugging in the figure. The resulting changes, ch1, ch2, ..., chn, are integrated into the program. This process can be repeated until all test cases pass. The novel component of this parallel-debugging process, Cluster, is shown in more detail in the

Figures above. We have developed two techniques to Cluster failed test cases. This section presents details of these techniques.

Drawback

Parallel programs are more difficult to debug than sequential programs due to their nondeterministic behavior. Nondeterministic nature of parallel programs is the major difficulty in debugging. Order-replay, a technique to solve this problem, is widely used because of its small overhead. It has, however, several serious drawbacks: all processes of the parallel program have to participate in replay even when some of them are clearly not involved with the bug; and the programmer cannot stop the process being debugged at an arbitrary point. The user cannot stop parallel processes being debugged in an arbitrary manner. For example, assume that there are two communicating processes; the process Pa sends a message M at the event Sa and Pb receives it at Rb. Also assume that we found an erroneous behavior of Pb and tried to detect its cause by inserting breakpoints in Pb. Then we examined the behavior of Pb in detail to find that the real cause is in the message M and try to know why and how Pa generated the erroneous message. At this point, however, we will realize that Pa has already completed its erroneous procedure to generate M and has proceeded too ahead to examine the cause of the error, because Order-replay execution must obey the causality of the events Sa and Rb. That is, it is impossible to stop Pa and Pb as we wish, at a point before Sa and a point after Sb. It is possible to rerun the program by Order-reply mechanism inserting a breakpoint at a point before Sa, but such a overrun-and-rerun debugging is seriously inefficient. Another defect is that all processes of the parallel program have to participate in replay execution even when some of them are clearly not involved with the bug. The processes irrelevant to the bug are not only obstructive to debugging but also wasteful of expensive computing resource. Since a modern parallel computer may consist of hundreds or thousands of processors, this problem becomes more serious.

3.2.1.B *Runtime Debugging Technique*

A runtime debugger actually modifies your code so that at runtime every memory reference is checked for validity. The debugger keeps track of your program's memory and immediately complains if you misuse a pointer, overwrite memory, free a block too many times, use an un-initialized variable, and more.

Furthermore, it does this to your entire program, regardless of where you think bugs might be found. It also doesn't make the simple errors that humans do, which often introduces more problems while looking for something else entirely!
Many runtime debuggers only check dynamically allocated memory blocks, which means that code would pass with no error found.

It locates the error by traversing each line of code and allows fixing it in a short time. A run-time error is a semantic error. The code checked-out fine by the compiler, but when the program executed something ran amuck. There is an error in the meaning, or semantics, of the program's behavior or in the logical flow of the program. Such errors (e.g., accessing a non-existing array element) are not caught by the compiler because it cannot anticipate changes in variables, and the effect of such, as a program executes. The symptoms for run-time errors are often strange and seemingly unrelated to the underlying problem in the code. Because of this, run-time errors are generally more difficult to correct than compile errors. This hits home at a point made earlier: Take it one step at a time. If you write a lot of code before testing, you are asking for trouble. There are two broad categories of run-time errors: those that cause the program to crash, and those that cause the program to generate incorrect results or to behave improperly. Both are semantic errors: the former are errors in how you used Java, the latter are errors in how you approached the problem.

What programmers needed was a tool that could execute one instruction of a program at a time, and print values of any variable in the program. This would free the programmer from having to decide ahead of time where to put print-statements, since it would be done as he stepped through the program. Thus, runtime debuggers were born. In principle, a runtime debugger is nothing more than an automatic print-statement. It allows the programmer to trace the program path and the variables without having to put print- statements in the code.

Today, virtually every compiler on the market comes with a runtime debugger. The debugger is implemented as a switch passed to the compiler during compilation of the program. Very often this switch is called the "-g" switch. The switch tells the compiler to build enough information into the executable to enable it to run with the runtime debugger.

The runtime debugger was a vast improvement over print statements, because it allowed the programmer to compile and run with a single compilation, rather than modifying the source and re-compiling as he tried to narrow down the error.

Drawback

Un-initialized memory errors might be incorrectly read and therefore are suppressed. It does not actually look at the source of bug but just the symptoms. Runtime debuggers made it easier to detect errors in the program, but they failed to find the cause of the errors. The programmer needed a better tool to locate and correct the software defect.

Software developers discovered that some classes of errors, such as memory corruption and memory leaks, could be detected automatically. This was a step forward for debugging techniques, because it automated the process of finding the bug. The tool would notify the developer of the error, and his job was to simply fix it.

Automatic debuggers come in several varieties. The simplest ones are just a library of functions that can be linked into a program. When the program executes and these functions are called, the debugger checks for memory corruption. If it finds this condition, it reports it. The weakness of such a tool is its inability to detect the point in the program where the memory corruption actually occurs. This happens because the debugger does not watch every instruction that the program executes, and is only able to detect a small number of errors.

3.2.1.C ***Holistic Debugging Technique***

It is a novel method for observing complex computer software running in instruction set simulators. A holistic debugger provides a translation framework that maps low-level data probed from the simulator to source-level application data. It also includes symbolic debuggers for inspecting individual processes in a simulated system. The debuggers are controlled by a debugger shepherd, which supports coherent observation of all participating processes in a distributed system. The shepherd is programmable and allows users to create new observation tools and debugging abstractions, and to write application-specific surveillance routines. A holistic debugger provides a translation framework that maps low-level data to source-level application data. A holistic debugger should not be thought of as yet another tool that solves a particular, narrow problem better than other tools. Although it can be used as an interactive debugger, its primary purpose is to serve as a meta-tool that enables construction of new tools, based on more robust techniques than existing tools.

Modern computers are in-deterministic. There are factors affecting program execution that cannot be accurately predicted, for example interrupt arrival times, memory communication interleaving, subroutine execution times, and clock readings. Complex programs are always affected by such random factors, and program executions are therefore not fully reproducible, unless the program is explicitly designed to be independent of unpredictable factors. In theory, repeatable execution is a prerequisite for the standard repetitive debugging procedure. In practice, repetitive debugging is meaningful for simple programs, as long as the variations are small. In-deterministic execution effectively prevents construction of scalable observation tools. Development and use of automated tools when experiments cannot be reliably reproduced is usually too time-consuming to be worthwhile. As an example, consider the multitude of debuggers for distributed software, using standard debuggers as building blocks. Although such debuggers have potential for debugging complex software, they have not become widely used. Due to in-deterministic factors, repeated executions of a particular program tend to differ at some points, for example in interleaving of events. Automated debuggers are generally not able to adapt to variations in an intelligent manner, and are therefore not practical for in-deterministic programs.

Any attempt to monitor a computer system with software probes will change the system's behavior. This is referred to as probe effect. The probe effect contributes to the indeterminism problem, and also limits the amount of data that can be observed in a running system. A monitoring service that suffers from probe effect provides a service whose quality degrades with increased usage. Such a service is inherently fragile and unsuitable for scalable observation tools.

There is no global clock in distributed systems, and a global ordering on all events in a system can often not be determined, even in post-mortem. Observing a partial ordering de_ned by the happens-before relation, however, is sufficient for observing the execution of a distributed system. In order for a tool to observe this partial ordering, it must be able to observe all messages sent between processes, and their points of arrival. This can be difficult in practice. In some distributed systems, messages and arrival points are straightforward to record, as in the case of network packets delivered to an application. Other types of messages, such as cache-to-cache transfers, are difficult to observe, and building tools that record and replay distributed executions involving such messages is hard.

Holistic debugging takes a complete system perspective on distributed system observation. A holistic debugger runs a distributed software system in a simulator and provides the user with means to examine all components in a system simultaneously, at any abstraction level higher than the simulator's.

*a)*__*Machine observation*__

A complete system simulator provides non-intrusive access to all system state visible to software. Thus, we can at any time stop the simulator, freeze time in the simulated world, and retrieve state data relevant for the application. Unlike standard debuggers, which use probing services supplied by the operating system to probe the state of running processes, the holistic debugger must use non-intrusive probing techniques, and cannot rely on operating system services. It probes the simulator for machine state, but the information retrieved is raw, binary information that has been transformed by compilers, virtual machines, and operating systems, and is no longer easily comprehensible to humans. In order to make this information useful for a programmer, it must be translated back to the abstraction level the programmer deals with, i.e. to variables and types in the programming languages used in the application.

b) <u>Abstraction stacks</u>

Each program in a computer system runs in a machine, which interprets the program instructions and updates machine state accordingly. The most basic machine is the physical machine, where instructions are interpreted by hardware, and machine state is stored in physical storage, such as memory, disk, and registers. Each machine has a set of instructions that programs can use, and programmers use a compiler to translate source code into the machine's instruction set. A physical computer usually runs only one program directly on the hardware, and in many cases, this program is an operating system. The operating system provides virtual machines, in which other programs can run. The programs in the virtual machines are likewise programmed in a high-level language, translated by a compiler to machine instructions. Some of these programs may in turn form other types of virtual machines, interpreting some program, which may be generated by a compiler, and so on. Computer systems generally contain a number of such abstraction stacks, seldom more than a few levels deep. For each program in a stack, there is a symbolic transformation, where a compiler transforms source code to machine code. There is often no straightforward way to perform the reverse translation from machine code to source code without help from the compiler, but most compilers are able to provide debugging information that contains adequate information to perform reverse translation, even in the presence of compiler optimizations. For each virtual machine in the stack, there is also a machine transformation; the storage of the program running in the virtual machine is mapped to storage in the machine that is running the program providing the virtual machine. For example, the virtual memory and registers of the virtual machine corresponding to a Unix process is mapped to physical registers, memory, or disk blocks. The machine transformation is usually reversible, if the state of the virtual machine can be examined.

c) ***Translation stacks***

In our design of a holistic debugger, for each inspected process in the simulated system, there is an associated abstraction translation stack. A translation stack consists of pairs of symbolic context objects and machine context objects, corresponding to the symbolic transformations and machines of the inspected process. The structure is shown in Figure below. When the user inspects a particular program, a translation stack is instantiated. It includes a symbolic context object, with a symbolic translator that lets the user inspect the execution and state of the program, similarly to a standard debugger. The symbolic translator probes the underlying machine context objects for program state data. Machine context objects that refer to a physical machine probe the simulator for simulated machine state. Machine context objects that refer to virtual machines, for example operating system processes, include a virtual machine translator (VMT) - a component that translates requests for virtual machine state to state requests to the underlying machine context object. In order for the VMT to perform storage reference translations, it probes the state of the program providing the virtual machine, using its symbolic context object. There are no fundamental problems stacking translators in this manner, as long as the necessary information for performing reverse translations is available. The stacked translator design enables translation of the information available in the underlying simulators to any abstraction level in the system.

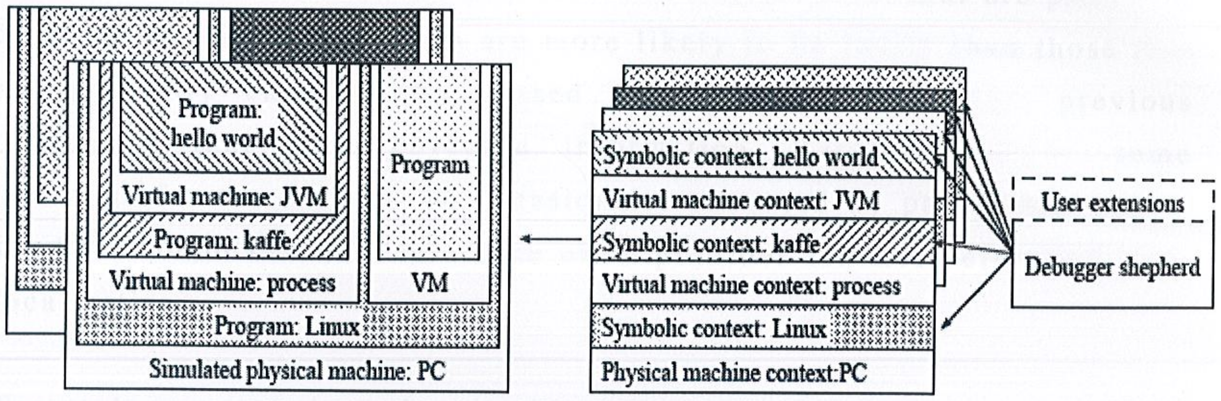


Figure 5: Working of Holistic Debugging

Holistic debugging addresses a major debugging problem that currently has no good solution. Nevertheless, we believe that, its most important potential is as a robust platform for building other software observation and analysis tools, much like an operating system is a solid platform for other programs.

Drawback

The main drawback of running multiple applications in a simulator is that it runs much slower than on a physical machine.

3.2.1.D *Tarantula Technique*

Software testers often gather large amounts of data about a software system under test. These data can be used to demonstrate the exhaustiveness of the testing, and find areas of the source code not executed by the test suite, thus prompting the need for additional test cases. These data can also provide information that can be useful for fault localization. Tarantula utilizes such information that is readily available from standard testing tools: the pass/fail information about each test case, the entities that were executed by each test case (e.g., statements, branches, methods), and the source code for the program under test. The intuition behind Tarantula is that entities in a program that are primarily executed by failed test cases are more likely to be faulty than those that are primarily executed by passed test cases. Unlike most previous techniques that used coverage information, Tarantula allows some tolerance for the fault to be occasionally executed by passed test cases. We have found that this tolerance often provides for more effective fault localization.

Tarantula is a technique that displays the results of running suites of tests against software systems. By showing what portions of the code are executed by passed and failed tests, the system helps people identify faults in their programs. Software testers often gather large amounts of data about a software system under test. These data can be used to demonstrate the exhaustiveness of the testing, and find areas of the source code not executed by the test suite, thus prompting the need for additional test cases. These data can also provide information that can be useful for fault localization.

Previously, we presented our Tarantula technique and a visualization tool that uses the technique for assigning a value for each program entity's likelihood of being faulty. We did this by specifying a color for each statement in the program. We utilize a color (or hue) spectrum from red to yellow to green to color each statement in the program under test. The intuition is that statements that are executed primarily by failed test cases and are thus, highly suspicious of being faulty, are colored red to denote \danger"; statements that are executed primarily by passed test cases and are thus, not likely to be faulty, are colored green to denote \safety"; and statements that are executed by a mixture of passed and failed test cases and thus, and do not lend themselves to suspicion or safety, are colored yellow to denote \caution."

In particular, the hue of a statement, s, is computed by the following equation:

$$hue(s) = \frac{\frac{passed(s)}{total\,passed}}{\frac{passed(s)}{total\,passed} + \frac{failed(s)}{total\,failed}}$$

Suspiciousness

The statements that are passed primarily by failed test cases are highly suspicious of being faulty are colored red to denote "danger" and the statements that are passed primarily by passed test cases are not suspicious of being faulty are colored green to denote "safety". The statements that are executed by a mixture of failed and passed test cases do not lend themselves to suspicion or safety is colored yellow to denote "caution". The suspiciousness of a coverage entity e with the following equation:

$$suspiciousness(s) = 1 - hue(s) = \frac{\frac{failed(s)}{total\,failed}}{\frac{passed(s)}{total\,passed} + \frac{failed(s)}{total\,failed}}$$

The Suspiciousness value varies from 0 to 1, where 1 is the most suspicious. and 0 is the least suspicious. In Equation 1, passed(s) is the number of passed test cases that executed statement s one or more times. Similarly, failed(s) is the number of failed test cases that executed statement s one or more times. Total passed and total failed are the total numbers of test cases that pass and fail, respectively, in the entire test suite. Note that if any of the denominators evaluate to zero, we assign zero to that fraction. Our Tarantula tool used the color model based on a spectrum from red to yellow to green. However, the resulting hue(s) can be scaled and shifted for other color models. Although we expressed these concepts in the form of statement coloring, they compute values that can be used without visualization. The hue(s) is used to express the likelihood that is faulty, or the suspiciousness of s. The hue(s) varies from 0 to 1 I 0 is the most suspicious and 1 is the least suspicious. To express this in a more intuitive manner where the value increases with the suspiciousness, we can either subtract it from 1, or can equivalently replace the numerator with the ratio of the failed test cases for s. Also, note that we can define this metric for other coverage entities such as branches, functions, or classes.

Using the suspiciousness score, we sort the coverage entities of the program under test. The set of entities that have the highest suspiciousness value is the set of entities to be considered first by the programmer when looking for the fault. If, after examining these statements, the fault is not found, the remaining statements should be examined in the sorted order of the decreasing suspiciousness values. This specifies a ranking of entities in the program. For evaluation purposes, each set of entities at the same ranking level is given a rank number equal to the greatest number of statements that would need to be examined if the fault were the last statement in that rank to be examined. For example, if the initial set of entities is ten statements, then every statement in that set is considered to have a rank of 10.

		test cases							
		3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	suspiciousness	rank
	void main() {								
1:	int x,y,z;	1	1	1	1	1	1	0.5	7
2:	int m=z;	1	1	1	1	1	1	0.5	7
3:	if(y<z)	1	1	1	1	1	1	0.5	7
4:	if(x<y){	1	1	0	0	1	1	0.63	3
5:	m=y;}	0	1	0	0	0	0	0	13
6:	else if(x<z){	1	0	0	0	1	1	0.71	2
7:	m=y;}	1	0	0	0	0	1	0.83	1
8:	else	0	0	1	1	0	0	0	13
9:	if(x>y){	0	0	1	1	0	0	0	13
10:	m=y;}	0	0	1	0	0	0	0	13
11:	else if(x>z){	0	0	0	1	0	0	0	13
12:	m=x;}	0	0	0	0	0	0	0	13

Figure 6: Example of Tarantula Technique

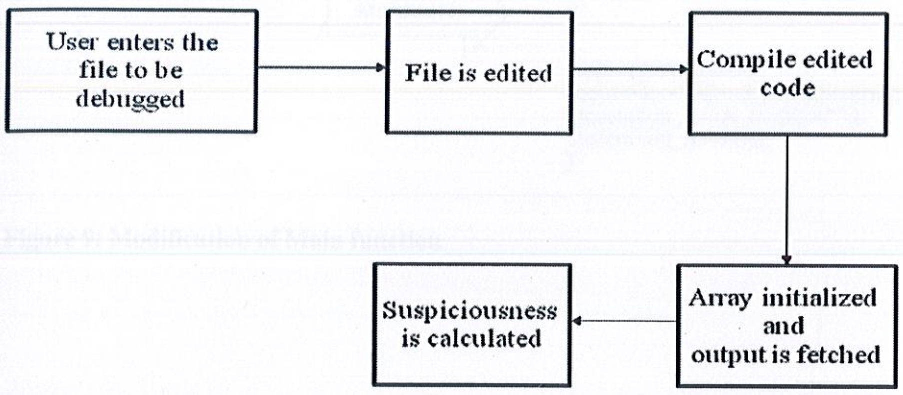
To illustrate how the Tarantula technique works, we provide a simple example program, mid(), and test suite, given in Figure 1. Program mid() takes three integers as input and outputs the median value. The program contains a fault on line 7|this line should read "m = x;". To the right of each line of code is a set of six test cases: their input is shown at the top of each column, their coverage is shown by the black dots, and their pass/fail status is shown at the bottom of the columns. To the right of the test case columns are two columns labeled "suspiciousness" and "rank."

The suspiciousness column shows the suspiciousness score that the technique computes for each statement. The ranking column shows the maximum number of statements that would have to be examined if that statement were the last statement of that particular suspiciousness level chosen for examination. The ranking is ordered on the suspiciousness, from the greatest score to the least score. Consider statement 1, which is executed by all six test cases containing both passing and failing test cases. The Tarantula technique assigns statement 1 a suspiciousness score of 0.5 because one failed test case executes it out of a total of one failing test case in the test suite (giving a ratio of 1). Using the suspiciousness equation specified in Equation 2, we get 1=(1 + 1), or 0:5. When Tarantula orders the statements according to suspiciousness, statement 7 is the only statement in the initial set of statements for the programmer to inspect. If the fault were not at line 7, she would continue her search by looking at the statements at the next ranks. There are three statements that have higher suspiciousness values than statement 1. However, because there are four statements that have a suspiciousness value of 0:5, Tarantula assigns every statement with that suspiciousness value a rank of 7 (3 statements examined before, and a maximum of 4 more to get to statement 1). Note that the faulty statement 7 is ranked first, this means that programmer would find the fault at the first statement that she examined.

CHAPTER-4

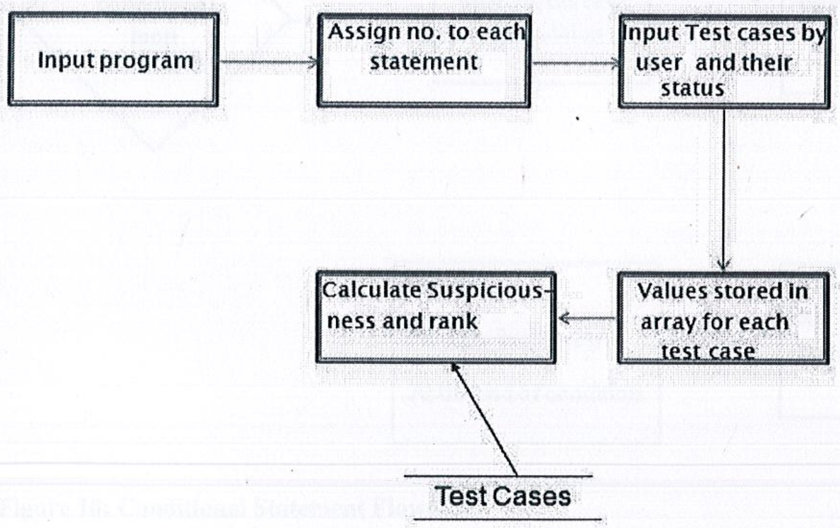
PROJECT DESIGN

4.1 Program Modules



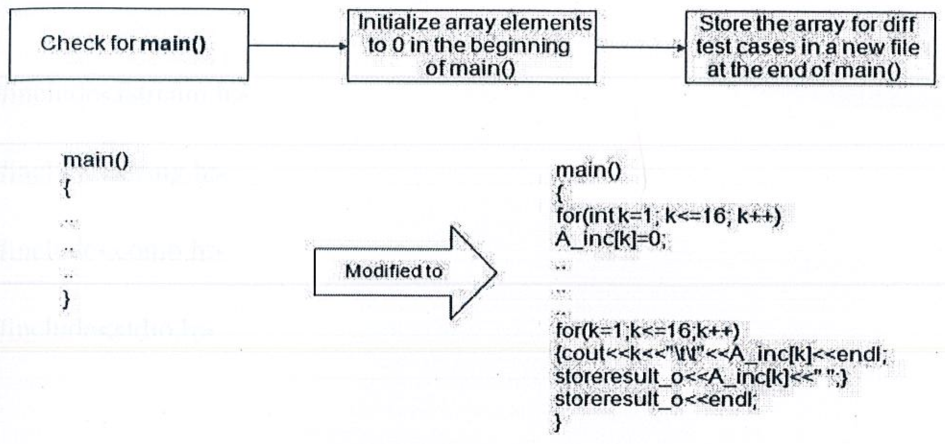
User enters the file to be debugged
File is edited
Compile edited code
Array initialized and output is fetched
Suspiciousness is calculated

Figure 7: Logic of Code



Input program
Assign no. to each statement
Input Test cases by user and their status
Values stored in array for each test case
Calculate Suspicious-ness and rank
Test Cases

Figure 8: Module Design


Check for **main()**
Initialize array elements to 0 in the beginning of main()
Store the array for diff test cases in a new file at the end of main()
Modified to

```
main()
{
..
..
..
}
```

```
main()
{
for(int k=1; k<=16; k++)
A_inc[k]=0;
..
..
..
for(k=1;k<=16;k++)
{cout<<k<<"\t\t"<<A_inc[k]<<endl;
storeresult_o<<A_inc[k]<<" ";}
storeresult_o<<endl;
}
```

Figure 9: Modification of Main function

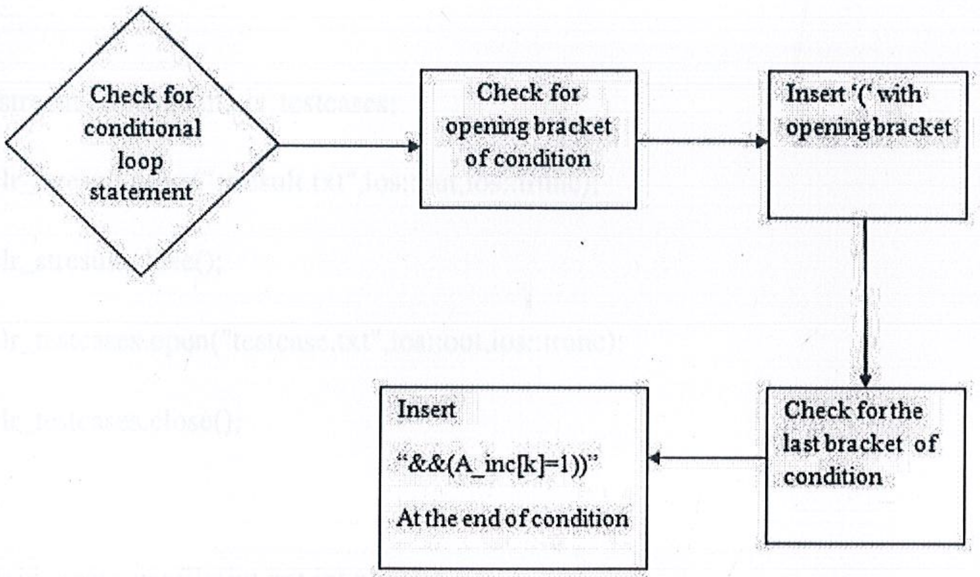

Check for conditional loop statement
Check for opening bracket of condition
Insert '(' with opening bracket
Check for the last bracket of condition
Insert
"&&(A_inc[k]=1))"
At the end of condition

Figure 10: Conditional Statement Flowchart

4.2 Source Code

```
#include<fstream.h>

#include<string.h>

#include<conio.h>

#include<stdio.h>


int count(ifstream &input);

void modify(int cnt,ifstream &input,ofstream &output);

void clear_files()

{

fstream clr_stresult,clr_testcases;

clr_stresult.open("stresult.txt",ios::out,ios::trunc);

clr_stresult.close();

clr_testcases.open("testcase.txt",ios::out,ios::trunc);

clr_testcases.close();

}

void createsuspfile(int cnt,int no_test_case)

{

ofstream susp;
```

```
susp.open("susp1_calc.cpp");

susp<<"\n#include<fstream.h>";

susp<<"\n#include<conio.h>";

susp<<"\nvoid main()";

susp<<"\n{";

susp<<"\nclrscr();";

susp<<"\nifstream input;";

susp<<"\nint array_value["<<no_test_case<<"]["<<cnt<<"],test_pass["<<no_test_case<<"],i,j;";

susp<<"\nint totalpass=0,totalfail=0,pass=0,fail=0;";

susp<<"\nint rank["<<cnt<<"][2],max,maxpos,max_temp,pos_temp;";

susp<<"\nfloat susp["<<cnt<<"],susp_cpy["<<cnt<<"][2],fd,pd;";

//int test_array[];

susp<<"\ninput.open(\"samp.cpp\");";

susp<<"\nfstream store_o;";

susp<<"\nfstream store_test;";

susp<<"\nfstream suspec;";

susp<<"\nstore_o.open(\"stresult.txt\",ios::in);";

susp<<"\nstore_test.open(\"testcase.txt\",ios::in);";

susp<<"\nsuspec.open(\"suspecious.txt\",ios::out);";

susp<<"\nchar ch;";

susp<<"\ninput.close();";
```

```
susp<<"\ni=0;";

susp<<"\ndo{";

susp<<"\nstore_o.get(ch);";

//cout<<ch;

susp<<"\nif(ch=='0')";

susp<<"\n{";

susp<<"\narray_value[i/"<<cnt<<"][i%"<<cnt<<"]=0;";

susp<<"\n}";

susp<<"\nelse";

susp<<"\nif(ch=='1')";

susp<<"\n{";

susp<<"\narray_value[i/"<<cnt<<"][i%"<<cnt<<"]=1;";

susp<<"\n}";

susp<<"\ni++;";

susp<<"\nif((i%"<<cnt<<")==0)";

susp<<"\n{";

susp<<"\nstore_o.get(ch);";

//    cout<<ch;

susp<<"\n}";

susp<<"\n}while(store_o);";

susp<<"\ni=0;";
```

```
susp<<"\ndo";
susp<<"\n{";
susp<<"\nstore_test.get(ch);";
susp<<"\ncout<<ch;";
susp<<"\nif(ch=='0')";
susp<<"\n{";
susp<<"\n      test_pass[i]=0;";
susp<<"\n      totalfail++;";
susp<<"\n      }";
susp<<"\nelse";
susp<<"\n      if(ch=='1')";
susp<<"\n      {";
susp<<"\n      test_pass[i]=1;";
susp<<"\n      totalpass++;";
susp<<"\n      }";
susp<<"\ni++;";
susp<<"\n}while(store_test);";
susp<<"\ncout<<\"\\n\\n\\n\\n\";";
susp<<"\nstore_o.close();";
susp<<"\n      for(i=0;i<"<<cnt<<";i++)";
susp<<"\n      {";
```

```
susp<<"\n       pass=0;";

susp<<"\n       fail=0;";

susp<<"\n               for(j=0;j<"<<no_test_case<<";j++)";

susp<<"\n               {";

susp<<"\n                       if(array_value[j][i]==1)";

susp<<"\n                       {";

susp<<"\n                               pass++;";

susp<<"\n                       }";

susp<<"\n                       if(array_value[j][i]==0)";

susp<<"\n                       {";

susp<<"\n                               fail++;";

susp<<"\n                       }";

susp<<"\n         cout<<array_value[j][i]<<\"  \";";

susp<<"\n               }";

//cout<<pass<<"  "<<fail<<"\n";

susp<<"\n               fd=fail/totalfail;";

susp<<"\n               pd=pass/totalpass;";

susp<<"\n               susp[i]=fd/(fd+pd);";

susp<<"\nsusp_cpy[i][0]=susp[i];";

susp<<"\nsusp_cpy[i][1]=i;";

susp<<"\n               cout<<\"       \"<<susp[i]<<\"\\n\";";
```

```
susp<<"\nsuspec<<susp[i]<<\"\\n\";";
susp<<"\n       }";
susp<<"\n       for(i=0;i<15;i++)";
susp<<"\n       {";
susp<<"\n              maxpos=i;";
susp<<"\n              max=susp_cpy[i][0];";
susp<<"\n              for(j=i+1;j<15;j++)";
susp<<"\n              {";
susp<<"\n                     if(susp_cpy[j][0]>max)";
susp<<"\n                     {";
susp<<"\n                            max=susp_cpy[j][0];";
susp<<"\n                            maxpos=j;";
susp<<"\n                     }";
susp<<"\n              }";
susp<<"\n              max_temp=susp_cpy[maxpos][0];";
susp<<"\n              susp_cpy[maxpos][0]=susp_cpy[i][0];";
susp<<"\n              susp_cpy[i][0]=max_temp;";
susp<<"\n              pos_temp=susp_cpy[maxpos][1];";
susp<<"\n              susp_cpy[maxpos][1]=susp_cpy[i][1];";
susp<<"\n              susp_cpy[i][1]=pos_temp;";
susp<<"\n       }";
```

```
susp<<"\n      cout<<\"\\n\\n\\nRank\\n\\n\";";
susp<<"\n      for(i=0;i<15;i++)";
susp<<"\n      {";
susp<<"\n              for(j=0;j<2;j++)";
susp<<"\n              {";
susp<<"\n                      cout<<susp_cpy[i][j];";
susp<<"\n                      cout<<\"\\t\\t\\t     \";";
susp<<"\n              }";
susp<<"\n              cout<<\"\\n\";";
susp<<"\n      }";
susp<<"\ngetch();";
susp<<"\n}";
susp.close();
}
int crtfile_notest()
{
int no_test_case;
cout<<"Enter the no. of test cases you want to enter : ";
cin>>no_test_case;
ofstream cases;
cases.open("no_cases.txt");
```

```
cases<<no_test_case;

return no_test_case;

}

void main()

{

clrscr();

ifstream input,input1;

ofstream output;

clear_files();

int cnt,no_test_case;

input.open("samp.cpp");

output.open("user_op.cpp");

cnt=count(input);

input.close();

input1.open("samp.cpp");

modify(cnt,input1,output);

input1.close();

output.close();

no_test_case=crtfile_notest();

//cout<<no_test_case;

createsuspfile(cnt,no_test_case);
```

```
//getch();

}

int count(ifstream &input)

{

char ch;

int count,count1=0,count2=0,count3=0;

input.get(ch);

do

{

if(ch==';')

{

count1++;

}

else

if(ch=='i')

{

input.get(ch);

if(ch=='f')

{

input.get(ch);

if(ch=='(')
{
count2++;
}
}
}
else
if(ch=='w')
{
input.get(ch);
if(ch=='h')
{
input.get(ch);
if(ch=='i')
{
input.get(ch);
if(ch=='l')
{
input.get(ch);
if(ch=='e')
{
input.get(ch);
if(ch=='(')

{

count3++;

}

}

}

}

}

}

input.get(ch);

}while(input);

count=count1+count2+count3;

return count;

}

void modify(int cnt,ifstream &input,ofstream &output)

{

//rewind(input);

char ch,str[200];

int br_op,br_cl,main_br=0,k=0,main_strt=0,main_br_strt=0,main_arr_init;

int return_chek=0,return_str=0,str_i,str_n,main_disp=1;

//int comment=0;
```

```
input.get(ch);

//cnt=17;

output<<"int A_inc["<<cnt<<"];"<<endl;

/*output<<"class class_for_array"<<endl;

output<<"{"<<endl;

output<<"public:"<<endl;

output<<"int x["<<cnt<<"];"<<endl;

output<<"};"<<endl;

*/

output<<"#include<fstream.h>"<<endl;

output<<"#include<ctype.h>"<<endl;

do

{

/*if(ch=='/')

{

output<<ch;

input.get(ch);

if(ch=='/')

{

cout<<"hehe";

output<<ch;
```

```
input.fputs(str);

comment=1;

//cout<<str;

}

}*/

if(ch=='m')

{

output<<ch;

input.get(ch);

if(ch=='a')

{

output<<ch;

input.get(ch);

if(ch=='i')

{

output<<ch;

input.get(ch);

if(ch=='n')

{

output<<ch;

input.get(ch);
if(ch=='(')

{

output<<ch;

input.get(ch);

main_strt=1;

main_arr_init=1;

}

}

}

}

}

br_op=0;

br_cl=0;

if(ch==';')

{

output<<ch<<endl<<"A_inc["<<k<<"]=1;"<<endl;

k++;

//output<<"A_inc["<<k+1<<"]=0;"<<endl;

input.get(ch);

}

else
```

```
if(ch=='i')
{
output<<ch;
input.get(ch);
if(ch=='f')
{
output<<ch;
input.get(ch);
if(ch=='(')
{
output<<ch<<ch;
input.get(ch);
br_op=1;
br_cl=0;
do
{
if(ch=='(')
{
br_op++;
}
else
if(ch==')')
{
br_cl++;
}
output<<ch;
input.get(ch);
}while((br_op!=br_cl)&&(ch!='\n'));
output<<"&&(A_inc["<<k<<"]=1))";
k++;
}
}
}
else
if(ch=='w')
{
output<<ch;
input.get(ch);
if(ch=='h')
{
output<<ch;
input.get(ch);
```

```
if(ch=='i')
{
output<<ch;
input.get(ch);
if(ch=='l')
{
output<<ch;
input.get(ch);
if(ch=='e')
{
output<<ch;
input.get(ch);
if(ch=='(')
{
output<<ch<<ch;
input.get(ch);
br_op=1;
br_cl=0;
do
{
if(ch=='(')
{
br_op++;
}
else
if(ch==')')
{
br_cl++;
}
output<<ch;
input.get(ch);
}while((br_op!=br_cl)&&(ch!='\n'));
output<<"&&(A_inc["<<k<<"]=1))";
k++;
}
}
}
}
}
}
else
if(ch=='f')
```

```
{
output<<ch;
input.get(ch);
if(ch=='o')
{
output<<ch;
input.get(ch);
if(ch=='r')
{
output<<ch;
input.get(ch);
if(ch=='(')
{
output<<ch;
input.get(ch);
br_op=1;
br_cl=0;
do
{
if(ch=='(')
{
```

```
br_op++;
}
else
if(ch==')')
{
br_cl++;
}
output<<ch;
input.get(ch);
}while((br_op!=br_cl)&&(ch!='\n'));
}
}
}
}
else
{
if(main_strt==1)
{
if(ch=='{')
{
main_br++;
```

```
main_br_strt=1;

if(main_arr_init==1)

{

output<<ch;

input.get(ch);

//output<<"class_for_array class_for_array_obj;"<<endl;

output<<"clrscr();";

output<<" \nfstream check_test,write_test;";

output<<"\ncheck_test.open(\"no_cases.txt\",ios::in);";

output<<"\nint no_test_left=0;";

output<<"\nchar get_test_left;";

output<<"\ncheck_test.get(get_test_left);";

//output<<"\ncout<<get_test_left;";

output<<"\ndo";

output<<"\n{";

output<<"\n    no_test_left=no_test_left*10+toascii(get_test_left)-48;";

output<<"\n    check_test.get(get_test_left);";

output<<"\n}while(check_test);";

output<<"\ncheck_test.close();";

output<<"\nif(no_test_left!=0)";

output<<"\n{";
output<<"\nno_test_left--;";

output<<"\nwrite_test.open(\"no_cases.txt\",ios::out);";

output<<"\nwrite_test<<no_test_left;";

output<<"\nwrite_test.close();";

output<<endl<<"\nfor(int plmk0=0;plmk0<="<<cnt-1<<";plmk0++)"<<endl;

output<<"\nA_inc[plmk0]=0;"<<endl;

output<<" fstream
store_o;"<<endl<<"store_o.open(\"stresult.txt\",ios::in|ios::out|ios::app);"<<endl;

output<<" fstream
store_test;"<<endl<<"store_test.open(\"testcase.txt\",ios::in|ios::out|ios::app);"<<endl;

output<<"int test_case_value,test_case_value_again=0;";

main_arr_init=0;

}

}

else

if(ch=='}')

{

main_br--;

}

}

str_n=0;str_i=0;

if((main_strt==1)&&(main_br_strt==1)&&(ch=='r'))
```

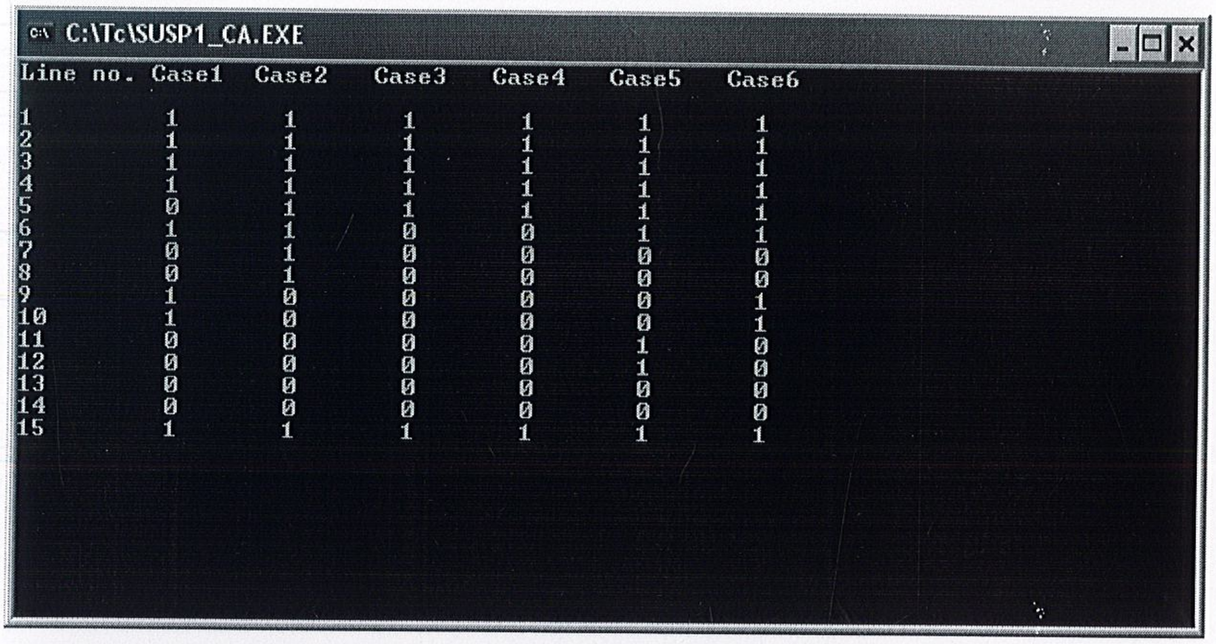
```
{
input.get(ch);
str[0]='r';
str_n=1;
return_str=1;
if(ch=='e')
{
str_n=2;
input.get(ch);
str[1]='e';
if(ch=='t')
{
str_n=3;
input.get(ch);
str[2]='t';
if(ch=='u')
{
str_n=4;
input.get(ch);
str[3]='u';
if(ch=='r')
{
str_n=5;
input.get(ch);
str[4]='r';
if(ch=='n')
{
return_str=0;
input.get(ch);
str[5]='n';
return_chek=1;
main_disp=0;
}
}
}
}
}
}
if (return_str==1)
{
for(str_i=0;str_i<str_n;str_i++)
output<<str[str_i];
```

```
return_str=0;

}

if(return_chek==1)

{

output<<"do{"<<endl;

output<<"if(test_case_value_again==1)"<<endl;

output<<"{clrscr();";

output<<"cout<<\"You have entered an invalid value.\"<<endl<<endl;"<<endl<<endl;

output<<"cout<<\"Please enter the valid value\"<<endl<<endl;}"<<endl<<endl;

output<<"cout<<\"Enter 1 or 0 for the test case is a pass or a fail respectively : \";"<<endl;

output<<"cin>>test_case_value;"<<endl;

output<<"test_case_value_again=1;"<<endl;

output<<"}while(test_case_value!=0&&test_case_value!=1);"<<endl;

output<<endl<<"for(int plmk=0;plmk<="<<cnt-1<<";plmk++)"<<endl;

output<<"store_o<<A_inc[plmk];";//<<\" \"";

output<<"store_o<<\"\n\"";

output<<"store_o.close();";

output<<endl<<"store_test<<test_case_value<<endl;"<<endl;

output<<"store_test.close();}"<<endl;

output<<"\nelse\n";

output<<"cout<<\"\\nYou have entered all the test cases\";";
```

```
output<<"getch();";
output<<"return ";
return_chek=0;
}
if((main_strt==1)&&(main_br_strt==1)&&(main_br==0)&&(main_disp==1))
{
output<<"do{"<<endl;
output<<"if(test_case_value_again==1)"<<endl;
output<<"{clrscr();"<<endl;
output<<"cout<<\"You have entered an invalid value.\"<<endl<<endl;"<<endl<<endl;
output<<"cout<<\"Please enter the valid value.\"<<endl<<endl;}"<<endl<<endl;
//output<<"cout<<\"Enter the valid value\"<<endl;"<<endl;
output<<"cout<<\"Enter 1 or 0 for the test case is a pass or a fail respectively : \";"<<endl;
output<<"cin>>test_case_value;"<<endl;
output<<"test_case_value_again=1;"<<endl;
//output<<"cout<<test_case_value;"<<endl;
output<<"}while(test_case_value!=0&&test_case_value!=1);"<<endl;
//output<<"cout<<test_case_value;"<<endl;
output<<endl<<"for(int plmk=0;plmk<="<<cnt-1<<";plmk++)"<<endl;
output<<"store_o<<A_inc[plmk];";//<<\" \";}";
output<<endl<<"store_o<<\"\\n\";"<<endl;
```

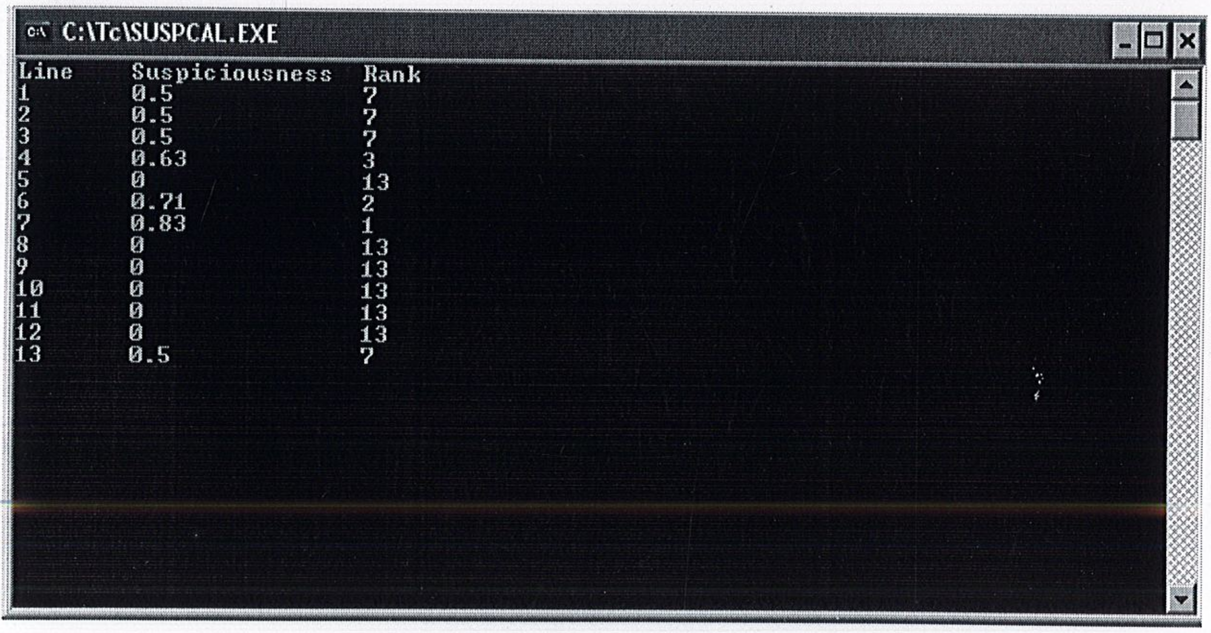
```
output<<"store_o.close();";

output<<endl<<"store_test<<test_case_value<<endl;"<<endl;

output<<"store_test.close();}"<<endl;

output<<"\nelse\n";

output<<"cout<<\"\\nYou have entered all the test cases\";";

output<<"getch();";

//output<<"storeresult_o<<A_inc[plmk]<<\" \";}"<<endl;

//output<<"storeresult_o<<endl;";

//output<<"storeresult_o.read((char *) & bo,sizeof (bo));"<<endl;

}

output<<ch;

input.get(ch);

}

}while(input);

}
```

4.3 ScreenShots



```
Line no. Case1   Case2    Case3    Case4    Case5    Case6

1          1       1        1        1        1        1
2          1       1        1        1        1        1
3          1       1        1        1        1        1
4          1       1        1        1        1        1
5          0       1        1        1        1        1
6          1       1        0        0        1        1
7          0       1        0        0        0        0
8          0       1        0        0        0        0
9          1       0        0        0        0        1
10         1       0        0        0        0        1
11         0       0        0        0        1        0
12         0       0        0        0        1        0
13         0       0        0        0        0        0
14         0       0        0        0        0        0
15         1       1        1        1        1        1
```

Figure 11: Values Corresponding To Each Line of Code



```
Line      Suspiciousness    Rank
1         0.5               7
2         0.5               7
3         0.5               7
4         0.63              3
5         0                 13
6         0.71              2
7         0.83              1
8         0                 13
9         0                 13
10        0                 13
11        0                 13
12        0                 13
13        0.5               7
```

Figure 12: Value of suspiciousness and rank corresponding to each line of code

4.4 Hardware/Software Requirements

- Windows platform
- At least 64 Mb of RAM

CONCLUSION

The tool is designed to identify the location in the code where there is a maximum probability of having an error. After the user inputs the program to be debugged, the tool passes the user entered test cases following which the tool assigns the suspiciousness value depending upon whether it has passed the test case or not and the line is executed or not. Higher the suspiciousness value more is the probability of having an error in that line of code. However, the tool works for the program having conditional statements only.

BIBLIOGRAPHY

- Nicholas Mc Guire *Runtime debugging in embedded systems,* Distributed & Embedded Systems Lab 2006
- Lars Albertsson , *Holistic debugging*. ISRN:SICS-T.2006.
- Gerard Ferrand and Alexandre Tessier, *Declarative Debugging.* INRIA.2005.
- James A. Jones, James F. Bowring and Mary Jean Harrold, *Debugging in Parallel.* 2007.
- James A. Jones and Mary Jean Harrold, *Tarantula Technique.* 2007.
- Yu-Chin Hsu, Bassam Tabbara, Yirng-An Chen and Furshing Tsai, *Advanced Techniques for RTL Debugging*. Novas Software Inc., 2025. 2006.
- A.P. Kryukov and AYa.Rodionov, *Dynamic-Debugging System for The Reduce Programs.*2007.
- Michiel Ronsse, Mark Christiaens, and Koen De Bosschere, *Cyclic Debugging Using Execution Replay.* ELIS Department.2005.
- Holger Cleve and Andreas Zeller, *Locating Causes of Program Failures*. 2008.
- www.ieeexplore.com
- www.wikipedia.com