A SOFTWARE FAULT INJECTION TOOL

A Project

Presented to the faculty of the Department of Computer Science

California State University, Sacramento

Submitted in partial satisfaction of
the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Gautham Devalapalli

SPRING
2013
A SOFTWARE FAULT INJECTION TOOL

A Project

by

Gautham Devalapalli

Approved by:

_______________________________, Committee Chair
Dr. Ahmed Salem

_______________________________, Second Reader
Dr. Ying Jin

___________________________
Date
Student: Gautham Devalapalli

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the Project.

__________________________________________, Graduate Coordinator
Dr. Behnam Arad                              Date

Department of Computer Science
Abstract

of

A SOFTWARE FAULT INJECTION TOOL

by

Gautham Devalapalli

Software testing is a very important phase in the software development life cycle. The quality and reliability of software are two major concerns of software testing. Developers assume that there are hidden bugs in software products because it is improbable to encounter all error paths of a complex system in a controlled test environment. Software has been an integral part of safety and mission critical systems. Software fault injection can help achieve this criterion. Fault injection involves deliberately inducing faults into software systems to observe their behavior in real time. This helps in identification of dependencies and improves fault tolerance algorithms and mechanisms.

Compile time injection is an experimental-based approach to system’s reliability and resilience. This approach requires injecting faults into source code of system under test and examining its behavior during the system’s compilation phase. The goal of the project is to develop a software fault injection tool (C-SWFIT). This tool is capable of injecting faults during the pre-execution phase of a program/software and assessing fault injection effects. This tool is designed to be intuitive and easy to use by software developers looking to test software tolerance and resilience.

______________________________, Committee Chair
Dr. Ahmed Salem

Date
ACKNOWLEDGEMENTS

I would like to take this opportunity to thank all the people who have been involved in this project, without whose professional knowledge, guidance and encouragement, this project would not have been successfully completed.

I would like to profoundly thank my project advisors, Dr. Ahmed Salem and Dr. Ying Jin, for giving me an opportunity to work under their guidance. Their professional inputs, advice and corrections are the key ingredients that went into making this project a success.

I want to express my special thanks to my good friends Sathvik Nerupalli and Vindhya Sankapally, who came to my rescue by helping me when I needed ideas with different perspective and painstakingly reviewing my report countless times.

Finally, I would like to thank my awesome parents who motivated me during every step in achieving my Master’s degree. My achievements are incomplete without their blessings and emotional support.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Acknowledgements</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>x</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xi</td>
</tr>
</tbody>
</table>

Chapter

1 INTRODUCTION .................................................................................................................. 1

2 FAULT INJECTION ............................................................................................................ 3

  2.1 Physical Injection ........................................................................................................ 4

  2.2 Software Implemented Injection .................................................................................. 4

    2.2.1 Run-Time Fault injection ..................................................................................... 5

    2.2.2 Compile-Time Fault Injection .............................................................................. 5

  2.3 Why Software Fault Injection ..................................................................................... 6

    2.3.1 Ferrari ................................................................................................................... 7

    2.3.2 Ftape .................................................................................................................... 7

    2.3.3 Doctor .................................................................................................................. 8

    2.3.4 Xception .............................................................................................................. 8

    2.3.5 Jaca ...................................................................................................................... 9

vii
2.4 Proposed Tool ............................................................................................................. 9

3 REQUIREMENTS ANALYSIS ......................................................................................... 10

3.1 Compile-Time Injection .............................................................................................. 10

3.1.1 Code Mutation ........................................................................................................ 10

3.2 Compile-Time Errors .................................................................................................. 11

3.2.1 Identifier Expected .................................................................................................. 11

3.2.2 Dangling else error ................................................................................................. 12

3.2.3 Keyword Omission ................................................................................................. 13

3.2.4 Variable not declared ............................................................................................. 13

3.2.5 Expected Punctuation Marks .................................................................................. 15

3.2.6 Incomplete Statement ............................................................................................. 16

3.3 Software Requirements .............................................................................................. 16

4 TOOL DESIGN AND IMPLEMENTATION .................................................................... 18

4.1 General Steps in Fault Injection .................................................................................. 18

4.2 C-Swfit ...................................................................................................................... 20

4.2.1 Source Code Loader .............................................................................................. 22

4.2.2 Injector ................................................................................................................... 24

4.2.3 Compiler .................................................................................................................. 25
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 : Comparison of various fault injection methods [3]</td>
<td>3</td>
</tr>
<tr>
<td>2.2 : Advantages and Disadvantages of Software Based Injection</td>
<td>6</td>
</tr>
<tr>
<td>3.1 : Example of Illegal character/Identifier</td>
<td>12</td>
</tr>
<tr>
<td>3.2 : Example of Dangling else Error</td>
<td>12</td>
</tr>
<tr>
<td>3.3 : Example of Keyword Omission</td>
<td>13</td>
</tr>
<tr>
<td>3.4 : Example of variable not declared</td>
<td>15</td>
</tr>
<tr>
<td>3.5 : Example of Expected-Punctuation Mark Error</td>
<td>15</td>
</tr>
<tr>
<td>3.6 : Example of Incomplete Statement error</td>
<td>16</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 : Steps for Performing Fault Injection</td>
<td>18</td>
</tr>
<tr>
<td>4.2 : Modules of C-SWFIT</td>
<td>20</td>
</tr>
<tr>
<td>4.3 : Screenshot of Tool GUI</td>
<td>21</td>
</tr>
<tr>
<td>4.4 : Code Snippet of Source Code Loader</td>
<td>22</td>
</tr>
<tr>
<td>4.5 : Loading an Invalid File</td>
<td>23</td>
</tr>
<tr>
<td>4.6 : Loading a Valid File</td>
<td>23</td>
</tr>
<tr>
<td>4.7 : Code Snippet of Injector Module</td>
<td>24</td>
</tr>
<tr>
<td>4.8 : Screenshot after an Injection is Complete</td>
<td>25</td>
</tr>
<tr>
<td>4.9 : Compilation Error</td>
<td>26</td>
</tr>
<tr>
<td>4.10 : Compilation Successful</td>
<td>27</td>
</tr>
<tr>
<td>4.11 : Code Snippet of Compiler Module</td>
<td>28</td>
</tr>
<tr>
<td>4.12 : Logger Messages after File Load and Injection</td>
<td>29</td>
</tr>
<tr>
<td>4.13 : Editor in Action</td>
<td>30</td>
</tr>
<tr>
<td>4.14 : Standard Output Display using a C++ Compiler</td>
<td>31</td>
</tr>
<tr>
<td>4.15 : Standard Output Display using a JAVA Compiler</td>
<td>32</td>
</tr>
<tr>
<td>4.16 : Successful Run Scenario</td>
<td>33</td>
</tr>
<tr>
<td>5.1 : Code Snippet of Character Fault Scenario</td>
<td>35</td>
</tr>
<tr>
<td>5.2 : Screenshot of Character Injection</td>
<td>35</td>
</tr>
<tr>
<td>5.3 : Screenshot of Dangling Else Injection</td>
<td>36</td>
</tr>
</tbody>
</table>
5.4 : Code Snippet of Keyword Replacement Fault Scenario ......................................... 37
5.5 : Screenshot of Keyword Replacement Injection.................................................. 38
5.6 : Output for Undeclared Variable Injection in a JAVA Program ............................. 39
5.7 : Output of Undeclared Variable Injection in a C++ Program................................. 40
5.8 : Code Snippet of Punctuation Mark Addition Fault Scenario ............................... 40
5.9 : Screenshot of Unexpected Braces Addition Injection ........................................ 41
5.10 : Code Snippet of Intentional Braces Removal Fault........................................... 42
5.11 : Screenshot of Intentional Punctuation Mark Removal ........................................ 43
CHAPTER 1

INTRODUCTION

Software testing is a key milestone in software engineering and a prerequisite of producing quality software. It involves execution of the code artifact produced at the end of the implementation phase and thus measures validity of the product built, verifies implementation of key functionality and ensures overall quality of the software being produced. There are multiple methodologies that define how this testing process is carried out, each at a different level of abstraction from the actual implementation.

There are many models being designed for software reliability. Software reliability has become an important criterion for software in the real world. Achieving 100% code coverage is not always possible and hence identification of all the bugs in the code is not always possible.

There are numerous testing methodologies, which can be implemented to identify a large amount of bugs. Fault injection testing is one of the best ways to identify the faulty state of the system/code. Fault injection testing methodologies can include hardware or software based injections. Using these fault injection techniques erroneous conditions are injected into the system to identify how the system behaves.

In this project, a fault injection tool has been designed and implemented. This tool has a capability to inject compile time errors (lexical, syntactical and semantic). The rest of the documentation has been organized as follows. Chapter 2 defines what fault injection means and explains about various fault injection tools. Chapter 3 highlights the software
fault injection technique used and various compile time errors implemented in the tool. Chapter 4 explains about the tool design and implementation. Chapter 5 provides more information about the fault scenarios. Finally, Chapter 6 concludes the documentation and Chapter 7 lists some future enhancements.
 CHAPTER 

 2

FAULT INJECTION

Fault injection relies on injecting faults/errors into the system to monitor its behavior when stressed in unconventional ways. We first need to hypothesize a fault and determine the impact of a fault when injected into the code [1]. This helps determine the tolerance of a system to artificial faults and gives a better understanding of how the system may behave in case of an unexpected error in real world. Fault injection helps uncover faults, in error handling code paths, which are very hard to test and is time consuming, in complex systems. Fault injection is broken down into two categories: Physical injection and Software implemented injection as shown in the Table 2.1.

Table 2.1: Comparison of various fault injection methods [3]

<table>
<thead>
<tr>
<th></th>
<th>Hardware</th>
<th></th>
<th>Software</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>With</td>
<td>Without</td>
<td>Compile</td>
<td>Run time</td>
</tr>
<tr>
<td>Cost</td>
<td>contact</td>
<td>contact</td>
<td>time</td>
<td></td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Perturbation</td>
<td>None</td>
<td>None</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Risk of damage</td>
<td>High</td>
<td>Low</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Monitoring time</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>resolution</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accessibility of fault</td>
<td>Chip pin</td>
<td>Chip internal</td>
<td>Register</td>
<td>Register memory I/O</td>
</tr>
<tr>
<td>injection points</td>
<td></td>
<td></td>
<td>memory</td>
<td>controller/port</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>software</td>
<td></td>
</tr>
<tr>
<td>Controllability</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Trigger</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
2.1 PHYSICAL INJECTION

Physical faults injection uses some physical hardware to inject errors into the targeted system. For example, injecting faults onto the pins of IC to evaluate the coverage of specific mechanisms [2]. Physical injection uses specialized hardware with contact or without contact to the targeted system. Hardware with contact method involves use of special hardware in direct physical contact providing electrical current changes to chip, examples are methods such as pin-level probes and sockets. Hardware without contact method uses some external injector, no physical contact with the target system, such as heavy ion radiation or electromagnetic interference to produce currents in the IC [3]. Physical injection is very accurate since we have better control over point of injection. Physical or hardware implemented injection is very expensive as the special hardware is built for a particular system and cannot be used for another system with similar specifications.

2.2 SOFTWARE IMPLEMENTED INJECTION

Software implemented fault injection is popular now a days for its simplicity and flexibility and inexpensiveness. They do not require specialized hardware and can even emulate hardware faults. The software fault injector layers between the target system and the operating system, which is not possible in case of hardware injection [3]. The software injector may support different applications. Some shortcomings with software fault injection are

1. The injector cannot access locations that are not accessible to the software.
2. The complexity of faults depends on the memory allocated to the software by the system.

3. The software may change the code in the original software causing perturbations in the workload running on the target system.

4. Even though the monitoring time is low, preparing the faults is very time consuming.

Software fault injection is broken down into two categories based on when the fault is injected: During run time or during compile time.

2.2.1 RUN-TIME FAULT INJECTION

Run-time injection method uses a trigger to inject a fault into the target system. A trigger can be a time-out or a trap/exception. A time-out refers to waiting for a pre-determined time and triggering a fault when the timer runs out. A trap refers to triggering a fault when a certain event takes place [3]. When this happens, the control shifts to an interrupt handler, which invokes the injector. Code insertion [3] method requires inserting instructions into the code so that fault injection takes place before a certain instructions. The insertion can happen during run-time and does not change the original code.

2.2.2 COMPILe-TIME FAULT INJECTION

Compile time injection requires modification of the original code before loading the program image and executing it for simulated faults. It does not require any additional software or hardware during run-time. Code mutation is a compile time technique, which involves modifying the original lines of code so that they contain faults, most commonly
used method for this kind of fault injection. Compile time injection cannot inject faults as the program runs and takes a lot time because various inputs require multiple executions. We discuss compile time injection in more detail in Chapter 2. Table 2.2 lists some advantages and disadvantages of software based injection.

Table 2.2 : Advantages and Disadvantages of Software Based Injection

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Does not require any specialized hardware</td>
<td>• Cannot inject faults into locations not accessible to software</td>
</tr>
<tr>
<td>• Can run near real time scenarios</td>
<td>• Controllability is very limited</td>
</tr>
<tr>
<td>• Simple and low cost for implementation</td>
<td>• Requires modification of original source code to inject faults</td>
</tr>
<tr>
<td>• Can add new class of faults later</td>
<td>• Near impossible to model permanent faults.</td>
</tr>
</tbody>
</table>

2.3 WHY SOFTWARE FAULT INJECTION

Software tolerance/reliability is a major standard for judging safety critical software. A faulty decimal point in the source code of Tritan rocket caused the launch sequence to fail and costing the US government $3 billion dollars [5]. Having knowledge of when and where a fault might occur is essential to figuring out the severity and frequency of a fault. Knowledge of all faults in software is not possible even with exhaustive testing because humans are prone to error and any missed input may cause a fault. Fault injection reduces the impact of anomalies by introducing artificial anomalies to see the tolerance of the targeted system. If a system cannot sustain under these manufactured faults then the
system will definitely fail if a real world fault occurs. Fault injection provides the extra layer of testing coverage need to uphold the standard of software reliability.

The tools required for software fault injection must be able to parse a program automatically and insert faults. Writing fault scenarios on the tools must not be more complex than writing the actual application. We discuss below some major tools currently available.

2.3.1 FERRARI

Ferrari (Fault and Error Automatic Real-Time Injection)[6] is a tool developed at University of Texas at Austin. It uses software traps to inject memory, bus and CPU faults. It is a run-time fault injection tool so it uses time-triggers and program counters. When a trap triggers, it modifies the contents of the register the program counter points at to emulate data corruptions. The faults injected can be transient or permanent faults such as an address line error, a data line error or a condition bit error. Ferrari can emulate hardware faults.

2.3.2 FTAPE

FTAPE (Fault Tolerance and Performance Evaluator) is an activity measurement tool used to inject faults in heavy stress conditions [7]. FTAPE can inject faults into CPU, memory and disk components. FTAPE has the capacity to inject based on the current workload of the target program. It can also emulate high-level stress on different parts such as CPU, memory or disk. For example, a bit flip will emulate a fault in the fault
tolerant computer. FTAPE uses stressing different sections the system for fault propagation to test the tolerance of a system.

2.3.3 DOCTOR

Doctor (Integrated Software Fault Injection Environment) [8] emulates the occurrence of faults in distributed applications. Doctor can inject memory faults, CPU faults and network communication faults in real time. It can use three different triggers: time-out triggers, traps and code insertion. Time-out triggers cause transient memory faults while inserting new code causes permanent CPU faults. Communication faults causes messages to delay, duplicate, or be lost in transition. The delay time can be determined or follow a random generation function. Doctor also collects reliability information. Although doctor provides various ways, it is not sufficient to test large-scale distributed systems such as grids. Crashes are not available for emulation in Doctor, which are main faults for grids.

2.3.4 XCEPTION

Xception [9] requires no modification of source code or insertion of traps. It uses the debugging features on modern processors to trigger fault injection. The fault scenarios are reproducible. Events such as opcode fetch, load, and store from/to a specified address can trigger faults. Xception uses a fault mask while injecting a fault into a target system. Xception compares the mask to memory/data/register and using bit-level operations such as stuck-at-zero, stuck-at-one, bit-flip and bridging, to change the bits that are set to one[6].
2.3.5 JACA

JACA is a software fault injection tool, developed by FAPESP in collaboration with University of Coimbra, written in JAVA [10]. It uses high-level meta-object protocol to inject faults into the target system. It can perform both high-level injections using methods and attributes of objects in JAVA program and low-level injections affecting assembly-language elements. JACA can run on any machine running JAVA virtual machine. A UI provides access to various tool functions. JACA can generate reports in .csv file format [10]. JACA can provide detailed information to observe the behavior during the execution of the faulty code.

2.4 PROPOSED TOOL

The software injection tools currently available in the market are commercial tools designed for complex systems. The main limitation is that they are designed for specific systems and are not portable. In this project, a software fault injection tool is designed and implemented targeted at small scale applications. The tool is language independent (currently C++, java are supported). It is useful for independent developers who want to check their application for fault tolerance against compile time errors. The simplicity and ease of use will help students and developers with test coverage. It checks the ability of their applications or programs to deal with unexpected compile time errors. This can help in planning to counter any bugs that may arise after deployment.
CHAPTER 3

REQUIREMENTS ANALYSIS

This chapter deals with compile time injection and the most common compile time errors. It summarizes system requirements and software required to build the compile-time fault injection tool.

3.1 COMPILETIME INJECTION

Compile time injection requires hard coding the fault into the original source code and then executing the instrumented code. No other software is required and it causes no disturbance during execution. It does not allow code insertion while the program executes. Since the faults are hardcoded, we can emulate hardware, software and transient faults. Transient faults are cause erroneous state in a system but do not do any lasting damage to the target program [11]. There are three ways to create software fault scenarios for fault injection: adding code, replacing code and deleting code from the original source code. The key approach for compile time injection is code mutation.

3.1.1 CODE MUTATION

Code mutation [1] refers changing or modifying a part of the original code to generate faults. This instrumented code can be any code format from binary object code to source code including assembly code [1]. Example of a code mutation can be as follows

Suppose the target system has the following line of code.

\[ X = X + 1; \]
This statement can mutate to:

\[ x = x + 5; \]

by replacing the original value, or to

\[ x = x + 1 + x; \]

by adding additional code, or can be deleted.

As long as the mutation changes the original value of variable \( x \), the mutations are valid. Code mutations generally cause compile time errors. We use code mutation approach in the fault injection tool developed for the project.

### 3.2 COMPILE-TIME ERRORS

In order for a program to become executable, the compiler must compile the source code to machine code. Compile time errors are errors that occur during this conversion. The following compile time errors are generic compile time errors. These errors are implemented in the fault injection tool developed for the project.

#### 3.2.1 IDENTIFIER EXPECTED

Identifier is an alternative name for a variable. Identifier name consists of letter, underscore and numbers. Any other characters will make the identifier invalid. Addition of special characters in the name of the identifier results in a compile time error known as Illegal character/Identifier Expected error. Some special character are “~!@#$%^&’” etc. Table 3.1 shows an example of an illegal character in the faulty code and the corrected code with no illegal character.
Table 3.1: Example of Illegal character/Identifier

<table>
<thead>
<tr>
<th>Faulty Code</th>
<th>Correct Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. private string 5numbers;</td>
<td>1. private string[ ]</td>
</tr>
<tr>
<td>2. public int k#k;</td>
<td>2. public int kk;</td>
</tr>
</tbody>
</table>

In the above code, in line 1, the variable of type ‘string’ should not start with a number. In line 2, there is a special character present in the name of the variable of type ‘int’.

The code has been corrected such that the variable name for ‘string’ does not have a number. Also, the special character of ‘#’ has been removed from the variable name of the data type ‘int’.

3.2.2 Dangling Else Error

Also known as else without if error. Conditional statements use logical conditions to flow of control in the code. In any programming language an ‘else’ statement is always associated with an “if” statement. Compiling the “else” block without an “if” block will lead to the fault. The compiler throws this error when it finds an “else” block without a preceding “if” block. Table 3.2 shows an example of a dangling “else” block in the faulty code. Every “else” block must have a preceding “if” block.

Table 3.2: Example of Dangling else Error

<table>
<thead>
<tr>
<th>Faulty code</th>
<th>Corrected code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. int s=5;</td>
<td>1. int s=5;</td>
</tr>
<tr>
<td>2. if (s &lt;10)</td>
<td>2. if (s &lt;10)</td>
</tr>
<tr>
<td>3. { s+=10;</td>
<td>3. { s+=10;</td>
</tr>
<tr>
<td>4. else{ s+=5}</td>
<td>4. }</td>
</tr>
<tr>
<td>5. }</td>
<td>5. else{ s+=5; }</td>
</tr>
</tbody>
</table>

In the above code, there is an else block without preceding if block

In the above code, an else block is placed after the if block
3.2.3 KEYWORD OMISSION

Keywords play a vital role in the syntax of any programming language. Java uses “class” or “Interface” keywords to declare object definitions. We usually encounter this fault if the object definition does not have the keyword. Deliberately removing the “class” or “interface” keyword in a Java program will cause the compiler to throw the “class/interface expected” error. Replacing the keyword also causes the same error and is a subset of this fault. Replacing keywords might not always throw an error. Table 3.3 shows the consequence of omitting a keyword in the faulty code.

Table 3.3 : Example of Keyword Omission

<table>
<thead>
<tr>
<th>Faulty code</th>
<th>Corrected code</th>
</tr>
</thead>
</table>
| 1. public Test1  
2. {  
3. public void Testmethod () {}  
4. } | 1. Public class Test1  
2. {  
3. Public void Testmethod () {}  
4. } |
| The keyword “class” is missing from the Object definition. | Adding “class” keyword corrects the code. We can also add interface keyword if we choose to call a method. |

3.2.4 VARIABLE NOT DECLARED

A Variable is accessible only within the scope where it is declared. If a variable is called outside its scope the compiler throws a “variable not declared” error. The scope of a method defines the boundaries of a variable declared in it. Exiting that method will render the variable useless. We can avoid this error by declaring the variable in the main class scope. This way the user can access the variable anywhere in the class scope. Table
3.4 shows an example of variable not declared within the scope of “main” class. Variables that are important should be defined globally to avoid this error.
Table 3.4: Example of variable not declared

<table>
<thead>
<tr>
<th>Faulty code</th>
<th>Corrected code</th>
</tr>
</thead>
<tbody>
<tr>
<td>int var1 = 17;</td>
<td>int var1 = 17;</td>
</tr>
<tr>
<td>if (true)</td>
<td>int var2=5;</td>
</tr>
<tr>
<td>{</td>
<td>if (true)</td>
</tr>
<tr>
<td>int var2+=10;</td>
<td>}</td>
</tr>
<tr>
<td>System.out.println(&quot;Var2&quot;+var2);</td>
<td>}</td>
</tr>
</tbody>
</table>

Var2 is declared in the “if” block so it is bound by the scope of the “if” block. System.out.println fails because it cannot access var2.

Since var2 is defined in the main class, the code executes without any errors. The program prints the value of var2.

3.2.5 EXPECTED PUNCTUATION MARKS

Punctuation marks are “:”, “;”, “(”,”)”, “{”,”}” etc. Incorrect use of punctuation marks such as not using a pair of braces or flower brackets and illegal usage of semi-colon or colon can lead to this error. It is very possible to avoid always because it depends on human skill to code efficiently. Table 3.5 shows an example of an unexpected punctuation mark showing up in the code. This error can be avoided by keeping track of punctuation mark pairs such as braces or brackets and ending the statements promptly.

Table 3.5: Example of Expected-Punctuation Mark Error

<table>
<thead>
<tr>
<th>Faulty code</th>
<th>Corrected code</th>
</tr>
</thead>
<tbody>
<tr>
<td>public static void main (String [] args)</td>
<td>public static void main(String[] args)</td>
</tr>
<tr>
<td>{</td>
<td>}</td>
</tr>
<tr>
<td>Test t1= new Test();</td>
<td>Test t1= new Test();</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

Compiler throws error because of unexpected “{“ at line 4. Removing the flower bracket can clear this error.
3.2.6 INCOMPLETE STATEMENT

Each line of code must have a purpose. This error appears when we only type part of statement leaving it meaningless. The compiler will throw a “not a statement” error. This fault occurs because of human error. Table 3.6 shows an example of an incomplete statement in the code. Thinking about the purpose of a line of code helps avoid this error.

Table 3.6: Example of Incomplete Statement error

<table>
<thead>
<tr>
<th>Faulty code</th>
<th>Corrected code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Int score[]= new int[1];</td>
<td>1. Int score[]= new int[1];</td>
</tr>
<tr>
<td>2. Int a;</td>
<td>2. Int a;</td>
</tr>
<tr>
<td>3. Score[0]= 5;</td>
<td>3. Score[0]= 5;</td>
</tr>
<tr>
<td>6. Score[1];</td>
<td>6. a = Score[1];</td>
</tr>
<tr>
<td>7. System.out.println(a);</td>
<td>7. System.out.println(a);</td>
</tr>
</tbody>
</table>

The compiler will throw an error at line 6 because it is an incomplete statement. The purpose of this statement is ambiguous. We could assign value of array “score[1]” to variable a. This would fix the error.

3.3 SOFTWARE REQUIREMENTS

The software injection tool is built for windows based systems. The necessary software required to implement the tool are as follows:

- Compilers: Borland C++ compiler version 5.5 is a popular freeware compiler used to test C++ programs in the tool. Java version 1.7 is a development kit used to test java programs in the tool.

- IDE: MS Visual Studio 2012 is an integrated development environment used to develop the graphical user interfaces for all windows based platforms.
- Language: C# is a simple object oriented programming language. It is well integrated into the ms visual studio via its visual C# language service.

- Operating System: Windows 7 OS is developed by Microsoft. The ide ms visual studio 2012 seamlessly integrates into the OS and provides the necessary environment to develop the tool.
CHAPTER 4

TOOL DESIGN AND IMPLEMENTATION

This chapter gives a high-level overview of the software fault injection tool (C-SWFIT) developed in this project. We discuss in depth the modules required for the C-SWFIT tool to function.

4.1 GENERAL STEPS IN FAULT INJECTION

Every fault scenario must follow a series of steps for successful fault injection. Skipping a step will result in incorrect fault scenarios and are invalid.

![Diagram showing the steps for performing fault injection](image)

Figure 4.1: Steps for Performing Fault Injection
The steps mentioned in Figure 4.1 are used as basis for injecting a fault in our fault injection tool. First, we obtain the original source code (C++/JAVA program) and load it into the tool for injection. Second, the user should then select the type of fault to inject into the original source code. Third, Building instrumentation of the code involves injecting the fault into a copy of the original source code and loading that copy to the compiler. Fourth, the logger checks the instrumented code for verification of fault. Fifth, we compile the instrumented code. Sixth, we check to see if the system can catch the fault injected and the messages thrown by the compiler if it did.

Instrumentation of code is an automated process. It is impractical to manually hand pick locations for injecting faults in large programs with thousands of lines of code. The tool should parse the program and inject the fault at a random location generated by a random generating function. Logger provides details and location of fault injected into the instrumented code. The compiler compiles the instrumented code and loads the executable/.class file into the run module of the tool. A GUI for the tool provides ease of use. An editor is included in the tool. This is useful when the user wants to check a fault injection manually. The editor can also show the location of the injected fault in the instrumented code. Since the focus is on compile time injection, we do not need additional control when the program executes.
4.2 C-SWFIT

Compile-time software fault injection tool is a fault injection tool written in C#, developed as part of this project. The GUI is generated using Microsoft visual studio 2012. The tool can inject compile time errors into a C++/JAVA program. The various modules of C-SWFIT in Figure 4.2 have different functions detailed in the next section.

The user can use the GUI to load the faults using the source-code loader module. Loading the file gives access to various faults that are available for injection into the file. User can choose to compile the original source file and run it without injecting any faults. The Injector module injects the user-selected fault and builds the instrumented code. We can use the Logger module to verify the location of the injected fault in the instrumented code. The Editor module helps the user manually check the injected fault in the code. The
Compiler module uses the file extension to choose the correct compiler to compile the instrumented code. The Error Viewer module shows the errors thrown by the Compiler module. The Run module executes the compiled code.

Figure 4.3 : Screenshot of Tool GUI

Figure 4.3 shows the GUI of the C-SWFIT tool. It shows the options available to the user to perform compile time injection. The modules of the C-SWFIT tool are as follows.

- Source code Loader
- Injector
- Compiler
- Logger
- Editor
- Error Viewer/Standard Output
- Run Module
4.2.1 SOURCE CODE LOADER

The source code loader loads the original source code into the tool. A code snippet of the source code loader module details the events while loading the program.

```csharp
private void LoadFile_Click(object sender, EventArgs e)
{
    if (!this.compileInjector.TryInitializeSourceFile(sourceFileName.Text))
    {
        MessageBox.Show("Please make sure the source file name is valid!");
    }
    else
    {
        sourceFileName.ReadOnly = true;
        InjectFault.Enabled = true;
        CompilingFileName.Text = sourceFileName.Text;
        LoadFile.Enabled = false;
    }
}
```

Figure 4.4: Code Snippet of Source Code Loader

Clicking the **LoadFile** button triggers the lines in Figure 4.4. It first checks to see if the source file has a valid file extension (.cpp or .JAVA). If this fails, it throws an invalid file message. When the user puts in a valid file, the Loader enables the Injector Button and gives Control to the Injector module. It then disables the **LoadFile** button so that the user does not accidentally load another file and overwrite the existing loaded file. It also loads the file to the compiler in case the user wishes to compile the original file and execute it. The screenshots in Figure 4.5 and 4.6 give an idea of how the tool reacts when a valid and an invalid file is loaded.
Figure 4.5 shows that loading a file with an invalid extension (.cp) throws an error message asking the user to verify if the source file name is valid.

Figure 4.6 : Loading a Valid File
Figure 4.6 shows loading a valid file disables the LoadFile button and transfers control to Injector module.

4.2.2 INJECTOR

The injector module gives the user the option to select the fault scenarios from a drop down menu. The injector module injects the selected fault into a copy of the source file and loads the new-instrumented code file to the compiler. We detail various fault scenarios in Chapter 4. The code snippet in Figure 4.7 explains the logic of the Injector module.

```
private void InjectFault_Click(object sender, EventArgs e)
{
    this.log.WriteLine("\n");
    CompilingFileName.Text = this.compileInjector.InjectFault(this.faultTypes.SelectedItem.ToString());
    runButton.Enabled = false;
    runFileName.ReadOnly = true;
}
```

Figure 4.7 : Code Snippet of Injector Module

Clicking the InjectFault button triggers the module to inject the selected fault and instrument a new code file and handing the control over to the compiler module. We can inject faults multiple times to the original file. It creates a new-instrumented file each time and names the file according to the date and time created. After every injection, the Injector loads the file to the compiler and details the injection in the log window. It also disables the run button so that the user does not accidentally run the code before
compilation. It stores all the instrumented files in a temporary folder on the user file system. By default, the tool creates a temporary work folder in C:\temp\WF. This is done so that the original source file is untouched.

![Fault Injector Screenshot](image)

Figure 4.8: Screenshot after an Injection is Complete

Figure 4.8 shows the status of tool after a successful fault injection. Note that after completion of an injection the compiler loads the new-instrumented file into the compile field.

### 4.2.3 COMPILER

The compiler module compiles any source file pointed by the path present in the compile field and throws any messages into the standard output module. Only a successful compilation will result in activation of the run module. The standard output generator shows any errors that occur during compilation. The Compiler module is language
The compiler can distinguish file formats according to the file extensions and choose the correct compiler (Borland C++ compiler version 5.5 for .cpp and JAVA compiler version 1.7 for .JAVA).

![Compilation Error]

Figure 4.9: Compilation Error

From Figure 4.9 we can see that compiling an instrumented file throws the errors in standard output and disables the Run module. The compiler window also shows the compiler being used. In this particular case the tool is using Borland C++ version 5.5 compiler to compile the source code.
When a compilation is successful as is the case in Figure 4.10, the standard output always shows a default compiler message and enables the Run button in the GUI. This feature is helpful when the user wants to execute the compiled code. The compiler message may vary based on the compiler being used by the tool and the language of the source code.
Choosing a compiler based on file extension is a core part of the compiler module as shown in the Figure 4.11 code snippet. The compiler module points to a Borland C++ compiler if it is .cpp and JAVA compiler version 1.7 if it is .Java. If the file extension is something other than these two then the tool throws “Unknown file type” message.
4.2.4 LOGGER

The Logger module logs any action taken by the Source code loader and the Injector module. A successful file load will throw a “file has been loaded!” message to the logger. The Logger window shows the details of any injection selected by the user. It also shows the location of the instrumented code file after injection by the Injector module.

![Logger Window](image.png)

Figure 4.12: Logger Messages after File Load and Injection

Figure 4.12 illustrates a “file has been loaded” message after a successful file load and also the details of injection (in this case A Character Injection) and the location of the instrumented code after the injection is complete.
4.2.5 EDITOR

The Editor module allows for manually editing the file pointed by the path present in the compile field. The file can be the original source file or an instrumented code. The user has the ability for manual injection of faults. This feature gives the user the flexibility to choose a location for fault injection in the code. It also allows for testing any fault scenario unavailable in the Injector module. The Editor module calls the notepad application in built into Windows OS.

![Image of Editor module]

Figure 4.13 : Editor in Action

From Figure 4.13 we can see that the editor gives user the ability to open an instrumented code and tweak the fault injections as desired.
4.2.6 ERROR VIEWER

The Error Viewer/Standard output shows the messages thrown by the compiler. Compilers throw error messages by parsing the program in various ways. The Standard output displays the messages according to the compiler currently used in tool.

Figure 4.14: Standard Output Display using a C++ Compiler

Figure 4.14 shows the output when the tool uses a C++ compiler to compile the instrumented code with a character injection scenario.
Figure 4.15 illustrates the error viewer output when the tool uses a java compiler to compile the instrumented code with a character injection at random position.

4.2.7 RUN MODULE

The Run module activates only after successful compilation of the program. Running a compiled code requires there be no errors during compilation. It is impractical to run an error-ridden code. The Run module activates the correct executor according to the file extension of the compiled file. The Run module has control only when all the errors are cleared and the compilation is successful. The Run module calls a specific executor based on the extension of the compiled code such as C++ compiler if it is .exe and Java compiler in case of a .class.
Figure 4.16: Successful Run Scenario

Figure 4.16 depicts a successful run scenario in which the tool transfers control to the Run module when compilation is successful. User can choose to run the file. Clicking the Run button triggers the tool to show the output of the executed program in a new window. The Run module executes the compiled code (.exe in case of C++ and .class in case of JAVA) to show the output.
CHAPTER 5

FAULT SCENARIOS

This chapter details the various fault scenarios used in C-Swift tool. It discusses each fault scenario using code snippets to explain the impact on the original code.

5.1 FAULT INJECTION SCENARIOS

The basis for implementing fault scenarios in C-SWFIT is compile time errors which have been discussed in Chapter 2. We need to add those errors, which are language independent (appear in all the languages used) into the tool. Since code mutation is the fault injection technique implemented, the following three methods are used to inject the fault.

- by adding code
- by replacing existing lines of code
- by deleting code

Fault scenarios for C-SWFIT use the above methods for compile time injection. In the following section, we discuss the implemented fault scenarios.

5.1.1 CHARACTER INJECTION

This fault scenario covers the illegal character/identifier expected error. We inject an illegal character into the existing lines of code to generate this fault. We take into consideration those characters accepted by any programming language. A random generator function injects a random character at a random location in the code.
Figure 5.1: Code Snippet of Character Fault Scenario

Figure 5.1 illustrates the random generation function code used to choose the characters. The characters are from the first 127 characters that include all numbers, alphabets and special characters in a programming language. The length of the illegal characters is set to 10. The tool will inject random characters with a max length of 10 at a random position.

```csharp
public static string generateRandomString(Random rand, int length)
{
    StringBuilder strB = new StringBuilder();
    for (int i = 0; i < length; i++)
    return strB.ToString();
}
```

Figure 5.2: Screenshot of Character Injection
In the screenshot from Figure 5.2 the tool injected the code with value “x%3!=~$p” at a random position 84 generated by the random generator function. This causes the compiler to throw several errors related to illegal character/identifier expected error.

5.1.2 DANGLINGELSE INJECTION

A dangling else is a conditional else without a conditional if block. This fault scenario covers the dangling else scenario. The compiler throws this kind of error when the source code contains an else loop without an “if” condition loop. The tool parses the source code and injects the else loop at a random location to inject the dangling else fault.

Figure 5.3 : Screenshot of Dangling Else Injection

Figure 5.3 illustrates the use of a JAVA code by the tool to show this error. After the tool injects the else loop the JAVA compiler throws an error in the standard output. This fault scenario comes under code insertion category.
5.1.3 KEYWORD REPLACEMENT INJECTION

This fault scenario covers the class or interface expected error. We replace a keyword in the code with some other keyword. This causes the tool to throw an error message while compiling the code. Sometimes the keywords replaced might not generate errors such as replacing \texttt{int} to \texttt{char} in small programs and compilation would be successful. However, the outcome would not be the same when compared to outcome of the original code.

Figure 5.4: Code Snippet of Keyword Replacement Fault Scenario

Code snippet in Figure 5.4 shows the most common keywords used while writing a program. Here we formed keyword pairs for use by the 	extit{Injector} module. The tool will parse the program and replace a random keyword with its counterpart in the corresponding pair. When this happens, the compiler throws a class or interface expected error.
From the screenshot in Figure 5.5 we can see that the tool replaced class keyword with struct keyword. This causes the compiler to throw the “class, interface or enum expected” error.

### 5.1.4 UNDECLARED VARIABLE/SYMBOL

This fault scenario covers the undeclared variable/symbol error. The tool introduces an undeclared variable into the main function at a random location. This causes the compiler to throw a symbol not found error. Even if variable is declared but not in the same scope this error occurs. Declaring a variable in the outer scope helps avoid this error. Both compilers throw an undeclared symbol error when tool injects this fault scenario and compiles it.
Figure 5.6: Output for Undeclared Variable Injection in a JAVA Program

From the screenshot in Figure 5.6 we see that the tool injects an uninitialized variable at a random position 282. JAVA compiler gives a better output message for this kind of fault injection as compared to C++ compiler output.

Figure 5.7 shows the error message when a Borland C++ compiler is used by the tool. Notice how the error message is unclear. This error message may vary based on the compiler used by the tool. If a commercial compiler such as an Intel C++ compiler is used, the tool will give a different error message.
This fault scenario covers the expected punctuation marks like ;{}() error and not a statement error. The tool parses the code and injects any of the above-mentioned punctuation marks at a random location. This causes the compiler to throw the error.

```csharp
1. char[] myCharacters = new char[] { ';', '{', ',', '}', '(', ')' };
2. char randomCharacter = myCharacters[this.rand.Next(myCharacters.Length)];
3. private string AddRandomCharacter(string code, char randomCharacter)
4. {
5.     intrandomIndex = this.rand.Next(code.Length);
6.     return code.Insert(randomIndex, "" + randomCharacter);
7. }
```

Figure 5.7: Output of Undeclared Variable Injection in a C++ Program

5.1.5 UNEXPECTED PUNCTUATION ADDITION INJECTION

Figure 5.8: Code Snippet of Punctuation Mark Addition Fault Scenario
The code snippet in Figure 5.8 details the process of adding a random punctuation mark at a random location in the code. The tool parses the code to a random location and adds the punctuation mark at that location. When the tool compiles the program, it causes compilation errors. They can be Expected punctuation mark error or not a statement error or both as shown below.

Figure 5.9: Screenshot of Unexpected Braces Addition Injection

Figure 5.9 shows the consequence of intentionally adding bracket in the original source code. Injecting a JAVA program with a ‘)’ causes the compiler to throw a not a statement error and a “;” expected error.
5.1.6 INTENTIONAL BRACES REMOVAL INJECTION.

This fault scenario is similar to the Punctuation Addition fault scenario. The tool parses the code at a random location for any punctuation marks. When it finds a punctuation mark, it deletes it to inject the fault into the code. If it is unable to find the punctuation mark in the code, the tool throws an “unable to find punctuation mark” error message.

```csharp
1. if (locations.Count <= 0)
2. {
3.     this.log.WriteLine("[1]: Unable to find the character [0] in the code.", ch,
4.         this.Name);
5.     return code;
6. }
7. else
8. {
9.     intrandNumber = this.rand.Next(locations.Count);
10.    intrandIndex = locations[randNumber];
11.    this.log.WriteLine("[0]: removed the {1} character at position {2}", this.Name,
12.        ch, randIndex);
13.    return code.Substring(0, randIndex) + code.Substring(randIndex + 1, code.Length
14.        - randIndex - 1));
15. }
```

Figure 5.10: Code Snippet of Intentional Braces Removal Fault

The code snippet in Figure 5.10 show the random function used to remove a punctuation mark from the code. It picks a random punctuation mark in between the beginning and ending of the main function and deletes it.
Figure 5.11: Screenshot of Intentional Punctuation Mark Removal

From the screenshot in Figure 5.11 we can see that the injector module removed a “;” from the source code at a random position 246 and compiled the instrumented code causing the error.
CHAPTER 6

CONCLUSION

Software resilience has become a critical criterion for quality software products. Ensuring software doesn’t fail due to unexpected faults is of greater importance than the software simply producing the correct output. Software fault injection aids in knowing whether the software meets this requirement and helps software engineers uncover hidden anomalies in software products. The focus of this project is compile time injection, an experimental-based approach. This approach has its limitations since it requires the user to hypothesize the faults. This approach provides no control during the execution phase. However, research has shown that analyzing the software behavior with known faults is an effective method in assessing system’s fault tolerance. Furthermore, software fault injection can be used to improve test coverage by a substantial margin.

Finally, a fault injection tool (C-SWFIT) was developed as part of this project. This language independent tool could successfully inject generic compile time faults in programs/software written in C++ or JAVA language. Using code mutation as basis, the tool is designed to inject faults based on user choice. This tool is useful for testing small to intermediate programs for fault tolerance.
CHAPTER 7

FUTURE ENHANCEMENTS

Future work includes upgrading the tool to:

- Inject faults during an application workload (run-time errors).
- Include support for every major programming language.
- Include report generation to record fault sessions for performance evaluation.
- Inject multiple faults simultaneously.
BIBLIOGRAPHY


