GATED SINGLE ASSIGNMENT FORM PARTNERED WITH VALUE-BASED STATISTICAL FAULT LOCALIZATION FOR NUMERICAL JAVA PROGRAMS

Oliver Traben

Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

Thesis Advisor: Dr. Andy Podgurski

Department of Computer and Data Sciences

Case Western Reserve University

May, 2023

GATED SINGLE ASSIGNMENT FORM PARTNERED WITH VALUE-BASED STATISTICAL FAULT LOCALIZATION FOR NUMERICAL JAVA PROGRAMS

Case Western Reserve University
Case School of Graduate Studies

We hereby approve the thesis[1] of

Oliver Traben

for the degree of

Master of Science

Dr. Andy Podgurski

Committee Chair, Advisor 4/18/2023
Department of Computer and Data Sciences

Dr. Vipin Chaudhary

Committee Member 4/18/2023
Department of Computer and Data Sciences

Dr. Harold Connamacher

Committee Member 4/18/2023
Department of Computer and Data Sciences

Dr. Michael Rabinovich

Committee Member 4/18/2023
Department of Computer and Data Sciences

[1]We certify that written approval has been obtained for any proprietary material contained therein.

Table of Contents

List of Tables

List of Figures

Acknowledgements

First of all, I would like to deeply thank my research advisor, Dr. Andy Podgurski. He introduced this thesis project to me, which was a fantastic opportunity, and ended up being a great interest to me. Also, Dr. Podgurski gave me continuous guidance and advice during the research process, which helped greatly in the advancement of my research. Additionally, I would like to thank Yigit Kucuk, who sacrificed some of his free time to assist me with many questions that I had related to his causal inference research. Lastly, I would like to thank my friends and family that have greatly supported me during my time here at Case Western Reserve University. Their words of encouragement always went a long way.

ABSTRACT

GATED SINGLE ASSIGNMENT FORM PARTNERED WITH VALUE-BASED STATISTICAL FAULT LOCALIZATION FOR NUMERICAL JAVA PROGRAMS

Oliver Traben

In recent years, new value-based statistical fault localization methodologies have been introduced that utilize causal inference techniques to better estimate where faults occur in a program. These methods (1) convert numerical Java programs into equivalent numerical Java programs in an inspired version of Gated Single Assignment (GSA) form, they (2) create causal maps that contain causal relationships between all of the GSA variables in the translated program, and (3) they use these causal relationships to adjust for confounding bias during the fault localization calculations. In this thesis, a new source-to-source compiler is introduced, and it converts numerical Java programs into true GSA form (not an "inspired" version like the previous studies). This new tool is tested on a set of numerical Java libraries and compared with the previous studies, and it is found that true GSA form causes a significant increase in fault localization correctness.

1 Introduction

1.1 Motivation

In the modern day, software is as important as it has ever been. Software has been adopted by nearly every aspect of modern society, which in turn has caused software to become increasingly complex and crucial to remain reliable. To ensure reliability within code, fault localization is used, which is the act of identifying the location of a defect or bug in a software system [22]. However, while fault localization is incredibly important to the software development process, it is often tedious and costly, especially as the size and complexity of the given software system increases. In some modern day software systems it may even be infeasible to manually locate software faults [22]. Thus, the software development field is in need of computational techniques of fault localization that involve minimal manual intervention. This type of fault localization has been labeled as automated software fault localization [13].

Within automated software fault localization, statistical fault localization (or spectrum-based fault localization) is often used as a way to deploy association strategies [13]. Using these statistical methods, execution profiles (or spectra) are collected from code, which display whether or not a given program statement or variable correspond to the occurrence of a program fault [13]. These execution profiles can then be used as a way to produce measures, called suspiciousness scores, which are used to quantify the likelihood that a given program statement or variable value will result in a software fault [13]. Once suspiciousness scores are collected for all program elements within a file of code, the scores can be compared and ranked, so then the software developers can examine the program elements with the highest ranks of suspicion first [13]; which will likely lead to a more efficient debugging process.

In this thesis, an existing statistical fault localization method called *CounterFault* will be used, which also utilizes causal inference techniques to attempt to eliminate confounding bias in the suspiciousness scores. *CounterFault* will be used alongside a newly developed source-to-source Java compiler that translates numerical Java programs to identical numerical Java programs in Gated Single Assignment (GSA) form. GSA form—in simple terms—is when all variables have at most one assignment statement. This allows *CounterFault* to excel, because each variable can be tracked to one statement, which helps *CounterFault* calculate the suspiciousness scores for each variable.

Although *CounterFault* has already been used in studies in the past, previous studies did not use a genuine GSA form. Rather, "inspired" versions of GSA form were used, which were claimed to be computationally equivalent to actual GSA form. These "inspired" GSA form tools led to the motivation for this thesis where a source-to-source Java compiler that utilizes a true form of GSA will be implemented. By utilizing this new tool with *CounterFault* and comparing it to previous studies, the impact of true GSA form can be analyzed as either more, less, or equally effective.

1.2 Related Work

In [6], Ding developed a tool that was intensely similar to the tool developed for this thesis. The tool converted programs into GSA form, then causal inference techniques were partnered with statistical fault localization to produce the suspiciousness scores for all of the variables within the program of interest [6]. However, the tool differed in that it was developed for numerical Python programs rather than numerical Java programs. Within the empirical study of Ding's research, the tool was able to correctly rank the faulty variable highly in most cases; but the tool was noticed to behave poorly when the faulty variable was faulty a high percentage of the time, or when it was faulty a low percentage of the time [6].

In [20], Sheng developed a tool that was very similar to Ding's tool. Nearly identical causal inference techniques and statistical fault localization methods were used. However, Sheng created the tool for numerical Java programs, similarly to this thesis. Also, Sheng's tool did not translate the Java program's into pure GSA form [20], like Ding's did. Instead, Sheng used a method that was inspired by GSA form, and was argued to result in the same variable outcomes [20]. Within the empirical experiment aspect of the study, the tool performed well; but it faced limitations when faced with faults not related to assignment statements [20].

In [19], Roach developed a tool which was basically the next iteration on Sheng's tool. Sheng's tool used Soot to convert the Java programs to the inspired-GSA form [20], but Roach's tool—similarly to this thesis—used ANTLR for the translation process [19]. However, Roach's tool still differed from this thesis in that it continued to use an inspired version of GSA form [19]. For the fault localization aspect of Roach's study, an existing statistical fault localization method called *CounterFault* [18] was used (just like for this thesis). *CounterFault* is more established than the fault localization methods used in the previous studies, so Roach's tool was able to rank the faulty variables highly—in terms of their suspiciousness scores—in most cases [19]. However, similarly to Ding's tool, the tool was noticed to behave poorly when the faulty variable was faulty a high percentage of the time, or when it was faulty a low percentage of the time [19].

In [14], Küçük utilized a modified version of Roach's tool for a highly in-depth analysis on value-based statistical fault localization for Java programs using causal inference techniques. Rather than using *CounterFault*, Küçük used a new and improved version of *CounterFault*, which was labeled as *UniVal* [13]. *CounterFault* only works on numerical and categorical assignment statements, whereas *UniVal* is able to handle numerical, categorical, boolean, structured types, and string assignment statements [14]. Due to these capabilities, Küçük was able to execute an extensive empirical study using a large Java library called Defects4J [5, 14]. Whereas in the other studies [6][20][19] only numerical programs could be considered.

1.3 Contributions

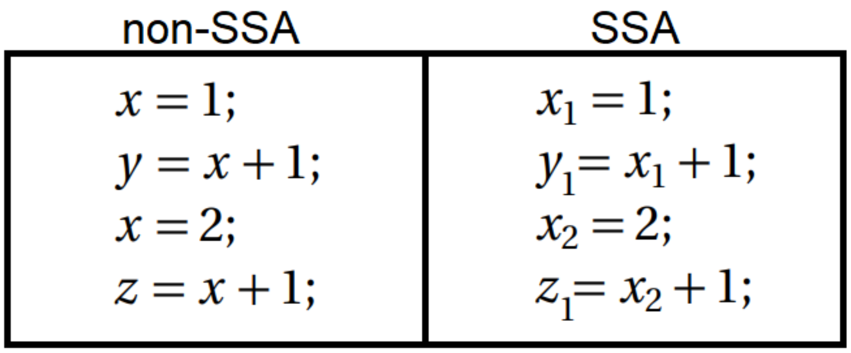
As described in the previous section, other than Ding's tool [6], all other previous studies [20][19][14] utilized an inspired-GSA form. Thus, it can be reasonably assumed that this may lead to some limitations. That is why, in this thesis, a source-to-source compiler will be used to convert Java programs into new Java programs that are in true GSA form. By no longer using an "inspired by" version of GSA, the causal inference strategies used during the statistical fault localization section of this study are expected to be more accurate.

Given that this is the first iteration of this tool, this thesis will compare more heavily to Sheng and Roach's studies. Meaning, this thesis will solely focus on numerical Java programs. By implementing a true GSA form translator for numerical Java programs, and testing it with CounterFault, this thesis's tool will be able to be cross-examined with previous tools, which will give a clear display of how much of a difference implementing true GSA form can make.

2 Background

2.1 Static Single Assignment Form

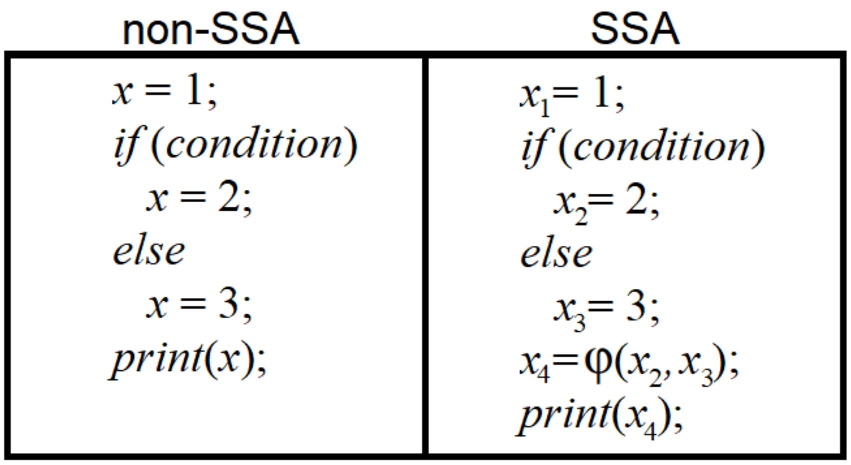
A program is in Static Single Assignment (SSA) form when every variable within the program has exactly one assignment statement [17]. As can be seen in Figure 2.1, when translating a program in non-SSA form to a program in SSA form, variables are all renamed to include an iteration number. Meaning, when a variable is assigned to a new value it is instead renamed to be the next iteration of that variable, so then every variable still only has one assignment statement associated with it (an example of this can be seen in Figure 2.1 with variable *x*).



non-SSA	SSA
$x = 1;$ $y = x + 1;$ $x = 2;$ $z = x + 1;$	$x_1 = 1;$ $y_1 = x_1 + 1;$ $x_2 = 2;$ $z_1 = x_2 + 1;$

Figure 2.1. Non-SSA vs. SSA

The SSA form described above works perfectly for programs with no conditionals. However, when conditionals are present—which is the case in most programs—programs split into multiple different branches, and this makes it difficult to determine what variable versions to use in future statements. To solve this issue, the ϕ-function is introduced (otherwise known as a pseudo-assignment function) which is designed to return the variable version from the branch that was actually selected during run-time [17]. For example, in Figure 2.2 the SSA code snippet shows that x_4 is assigned to $\phi(x_2, x_3)$, meaning when the $condition$ is true x_4 will be set equal to x_2, whereas when the $condition$ is false x_4 will be set equal to x_3. Therefore, the ϕ-function allows the program in Figure 2.2 to print the correct value at the end of the code snippet.



non-SSA	SSA

$x = 1;$
if (*condition*)
  $x = 2;$
else
  $x = 3;$
print(x);
	$x_1 = 1;$
	if (*condition*)
	  $x_2 = 2;$
	else
	  $x_3 = 3;$
	$x_4 = \varphi(x_2, x_3);$
	print(x_4);

Figure 2.2. Non-SSA vs. SSA (Conditionals)

Figure 2.2 makes the translation from a non-SSA conditional to a SSA conditional look fairly simple. However, when working with more complex conditional statements—particularly ones with more than two branches—the translation from non-SSA form to SSA form can get far more difficult. Plus, when working with loops even more difficulties get introduced. Thus, to deal with all of these issues, an extension of SSA is introduced called Gated Single Assignment (GSA) form. This extension of SSA will be described in detail in the following section.

2.2 Gated Single Assignment Form

SSA form is generally used to identify points in a program where variable definitions get uncertain (due to conditionals or loops), but it fails to actually interpret these points during runtime [17]. After all, the ϕ-function in SSA does not take a condition as input, so there is no way—programmatically—to determine the variable version at runtime. Therefore, Gated Single Assignment (GSA) form was introduced to replace all ϕ-functions with *gating functions* that can be executed at runtime [17]. In total, GSA has three distinct gating functions, which are described below:

- $\phi_{if}(p, v_1, v_2)$: this function is placed at the end of all if statements for each variable that is modified within the if statement. p represents the predicate of the if statement, v_1 represents the value that the given variable is equal to if the predicate evaluates to true, and v_2 represents the value that the given variable is equal to if the predicate evaluates to false [17]. Thus, this function has identical logic to an $if - then - else$ statement; because it just says that if p is true, return v_1, else, return v_2.
- $\phi_{entry}(v_{init}, v_{iter})$: this function is placed at the first usage of each variable within a loop. It's purpose is to determine whether the given variable should be set to it's initial value v_{init}, or to the value of the most recently iterated

version of the variable v_{iter} [17]. Thus, to put it simply, the function will return v_{init} until v_{iter} has been reached within the program.

- $\phi_{exit}(v_{init}, v_{exit})$: this function is placed at the end of every loop for each variable that is modified within that loop. v_{init} represents the value of the given variable before the loop is executed, and v_{exit} represents the value of the given variable if it were changed within the loop [17]. To put it simply, this function will only return v_{init} if the loop is executed and v is never iterated or reached within the execution of the loop; or, if the loop condition is never true to begin with.

2.3 Causal Inference

2.3.1 Causal Effects

Causal inference is a field of study that focuses on identifying and quantifying the causation of one variable onto another. The variable A that is being studied as the causation variable is known as a *treatment variable*, and the variable Y that is being measured in response to the treatment variable is known as the *outcome variable* [9]. For example, consider a theoretical scenario where a binary treatment variable A represents whether or not an individual has been vaccinated for some sickness, and a binary outcome variable Y represents whether or not an individual has died. To be more specific, $A = 1$ represents a subject that has been vaccinated, $A = 0$ represents an individual that has *not* been vaccinated, $Y = 1$ represents a subject that died, and $Y = 0$ represents a subject that did *not* die. Also, to clarify some more terminology, $Y^{a=1}$ and $Y^{a=0}$ are what we call *counterfactual outcomes*, which are the outcomes that are returned when $A = 1$ and $A = 0$, respectively [9]. Meaning, $Y^{a=1}$ would let us know whether or not a vaccinated subject died, and $Y^{a=0}$ would let us know whether or not a non-vaccinated subject died. Thus, now that we are familiar with the terminology, we can try to determine whether or not vaccination has a causal effect on death. The trivial way to determine this would be to observe the values of $Y^{a=1}$ and $Y^{a=0}$ and then compare them. If $Y^{a=1} = Y^{a=0}$ we would say A has no causal effect on Y, whereas if $Y^{a=1} \neq Y^{a=0}$ we would say A has a causal effect on Y.

The method described above sounds simple, but unfortunately—in virtually all real scenarios—a subject cannot receive both treatment values [9, 16]. Even if an individual received both treatment values, the second outcome would then be influenced by the first treatment value [16]. After all, when considering our example from above, we could take an untreated subject ($A = 0$) and measure their outcome Y, and—given that they did not die—we could then give them the vaccine ($A = 1$) and measure their outcome Y once again. However, the earlier treatment ($A = 0$) may now have an influence on the second treatment, which will impact the outcome Y. Plus, counterfactual outcomes are meant to represent the same subject under all of the same conditions (other than the treatment value itself). Thus, if Y is observed at two different times, any action taken between the two observations—or any other natural changes—will modify the conditions of the subject [16]. This issue is labeled as the *fundamental problem of causal inference*, because if we cannot measure both counterfactual outcomes $Y^{a=1}$ and $Y^{a=0}$ then it is impossible to measure whether or not a causal effect is present for an individual [16].

To bypass the fundamental problem of causal inference, the average causal effect in a population of subjects is often considered instead [9]. With an individual, we determined a causal effect was present if: $Y^{a=1} \neq Y^{a=0}$, whereas for an average causal effect, we say it is present if: $Pr[Y^{a=1} = 1] \neq Pr[Y^{a=0} = 1]$ [9]. To better understand this definition we will return to the vaccination example from earlier: if we have a population of individuals where some have been vaccinated, and some have not, we can measure the proportion of vaccinated individuals that have died and compare it to the proportion of non-vaccinated individuals that have died. If these proportions are unequal, we say that A has an average causal effect on the outcome Y.

Average causal effects help with the fundamental problem of causal inference, but they still are not perfect. After all, average causal effects are estimations of causal effects [9]. In order to truly calculate a causal effect, it is necessary to know all counterfactual outcomes for certain, which is generally impossible due to the fundamental problem of causal inference. Nonetheless, average causal effects are an improvement of individual causal effects, but because they are estimations, they leave room for biases [9]. Although there are many forms of biases, for this thesis, we will solely focus on confounding bias, which will be described in the following section.

2.3.2 Confounding Bias

Consider a study that measured data on people trying a new medication and people dying. The study discovers that people trying the new medication are more likely to die than the people who do not take the new medication. Does this imply that the new medication is *causing* people to die? Although that may seem like the logical explanation, it must be remembered that association does not always imply causation. For example, in this theoretical study it may be the case that people tend to only try the new medication when they are sick. Meaning that someone taking the new medication is a cause of sickness. Furthermore, it is known that death is also a cause of sickness. Thus, if sickness causes people to try a new medication and it causes people to die, is the medication really what is causing the people to die? Or is it the sickness? This interference from sickness is called confounding bias; and in this exact scenario, sickness would be labeled as a *confounding variable* or a *confounder* [9]. Figure 2.3 showcases this concept in a more general sense. L represents the confounding variable because it has a cause on both the treatment variable A and the outcome variable Y. Therefore, for a given subject from the example above, L would represent whether or not they are sick, A would represent whether or not they tried the new medication, and Y would represent whether or not they died.

$$\begin{tikzcd}
L \arrow[r] \arrow[rr, bend left] & A \arrow[r] & Y
\end{tikzcd}$$

Figure 2.3. Confounding Bias [9]

In this thesis, the confounding variables to identify and account for are program variables that may cause other program variables or statements to have high suspiciousness scores. For example, let's say there exists a faulty program variable F and another program variable N that has an assignment that depends on F. If this confounding bias is unaccounted for, N may be the variable that is given a high suspiciousness score instead of F, even though F is the variable that is truly responsible for N's faultiness. The strategies for locating and accounting for these confounding variables will be discussed in further detail later.

2.4 Source-to-Source Compilation

Source-to-source compilers (or transpilers) are special types of compilers that translate programs from one programming language to equivalent programs in a different programming language [12]. Generally, source-to-source compilers are used when converting a program to match the language of a given system, or when trying to convert useful code to a more optimized language [12]. However, in this thesis, source-to-source compilation will be used for statistical fault localization methodology. To do so, the regular approach to source-to-source compilers will be slightly modified. Instead of translating from one programming language to another, one programming language will be translated to a variant of itself that is in GSA form. Previous studies [20][19][14] also made source-to-source compilers to translate to GSA form. However, once again, the previous studies did not implement a true form of GSA. Thus, by implementing a source-to-source compiler that converts a program into a true form of GSA, causal inference methods can be used on our compiled programs to more accurately execute statistical fault localization.

2.5 Statistical Fault Localization

As briefly mentioned in the introduction of this thesis, statistical fault localization utilizes a variety of statistical methods to rank program elements by how likely they are to produce faults. To be more specific, statistical fault localization requires test inputs, which produce corresponding execution profiles [4]. These execution profiles are associated with each program element, and they keep track of whether or not that given program element was executed during the given test input. All of the execution profiles are compared to the output of the program, which must be labeled as a "pass" or a "fail" [4]. The type of statistical fault localization being described is coverage-based, and it generally concludes that those program elements that are executed more when a test input results in a "fail" implies that those program elements are more likely the cause of the fault [4]. The association between the execution profile of a program element and whether or not a test input resulted in a "fail" is used to calculate the *suspiciousness* score for that given program element [4]. All program elements are ranked by their suspiciousness scores in decreasing order, so then developers can easily go down the list of suspicious program elements until they discover the cause of the software fault [4].

Coverage-based statistical fault localization specifically excels in programs with lots of conditional statements [19]. This is due to the fact that suspiciousness scores are calculated solely off of whether or not a program statement was reached or not for each test case. Therefore, if a program had no conditional statements, every variable would be executed, and thus every suspiciousness score would be equal to 1. To account for this flaw, there exists another type of statistical fault localization called value-based statistical fault localization. In value-based statistical fault localization, the value of every program element will be recorded during each test execution [18]. These values are compared to each other for successful and unsuccessful program executions, so then suspiciousness scores can be calculated based off of variable values. Value-based statistical fault localization will be used for this thesis to avoid the limitations of pure coverage-based statistical fault localization techniques.

2.5.1 CounterFault

As mentioned earlier, this thesis will be utilizing an existing value-based statistical fault localization methodology, called *CounterFault*; which is particularly unique because it also utilizes causal inference techniques within it. *CounterFault* reads through the causal relationships of variables in a program, and uses these relationships to account for confounders [18].

The original *CounterFault* utilized an "inspired" version of GSA to gather the variable values (*value-profiles*) and their corresponding "parent" variables (i.e. the confounding adjustment variables) on each test program execution [18]. For a given test execution, once all of the value-profiles were gathered—along with the causal maps—each GSA variable would have a random forest regression model trained for it [18]. The random forest models were trained by inputting the value-profiles of the given variable, along with the values of the given variable's confounding variables [18]. For numeric variables, a set of 10 representative values would be derived from 10 equally spaced quantiles of the recorded values of the given variable [18]. Once these representative values were computed, each pair of representative values (a, b) would be used to compute two distinct counterfactual outcomes: $E[Y^{A=a}]$ and $E[Y^{A=b}]$ [18]. In *CounterFault*, the outcome variable Y represents whether the given program failed $(Y = 1)$ or passed $(Y = 0)$ [18]. On the other hand, the counterfactual outcomes $E[Y^{A=a}]$ and $E[Y^{A=b}]$ represent an estimate of how likely the program is to pass or fail, on a scale between 0 and 1 [18]. The pair (a, b) which results in the largest average causal effect will be assigned as the value of the given variable's suspiciousness score [18].

In this thesis, *CounterFault*'s statistical fault localization framework will be utilized to compute the suspiciousness score rankings. However, the previous GSA framework is being replaced by a new GSA source-to-source compiler that was developed for this thesis. This tool will be described in great detail in the chapter to follow.

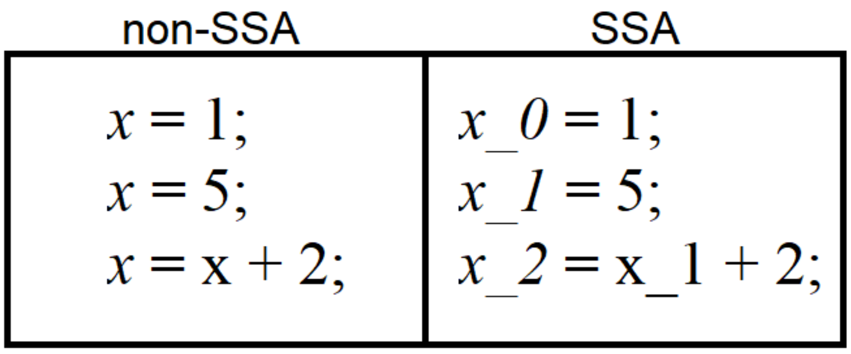
3 GSA Source-to-Source Compiler

3.1 Design

The GSA source-to-source compiler designed and implemented in this chapter is the fundamental component of this thesis. To summarize the compiler in simple terms: it takes a Java program, transforms it into an equivalent Java program in GSA form (as well as producing a causal map that contains each variable in the program), and then uses the GSA form file and the causal map to perform statistical fault localization. The specific design details will be elaborated further in the following subsections.

3.1.1 Single Static Assignment

Since GSA is an extension of SSA, the first design aspect to be focused on is generic SSA form. Just to reiterate, SSA form constraints that every variable should have exactly one assignment [17]. Thus, the design for this compiler would be to rename every single variable within the program so that the variable name includes a *variable-number*. The variable-number would start at 0 for each variable name and iterate up by one each time that variable name would be assigned to a new value. A hash map would be used to keep track of this variable-number, by having every variable name linked to a corresponding variable-number (which will be incremented each time that variable has a new assignment statement). As for the actual syntax, the variable-number would be appended to the end of each variable, but separated by an underscore (like the syntax shown in Figure 3.1).



non-SSA	SSA
x = 1;	x_0 = 1;
x = 5;	x_1 = 5;
x = x + 2;	x_2 = x_1 + 2;

Figure 3.1. Non-SSA Code Translated into SSA Code

3.1.2 If Statements

The first gating function focused on would be the ϕ_{if} function. As explained in section 2.2, a ϕ_{if} function needs to be placed at the end of every *if* statement chain for *each* variable that is changed within that given *if* statement chain. The design for tracking which variables were modified within a given *if* statement chain would be accomplished by saving a copy of the variable-number hash map at the time of entrance to the *if* statement chain, then comparing it with the variable-number hash map at time of exiting the *if* statement chain. Whichever variables would have variable-number values that differ between the two hash maps must have been modified somewhere within the *if* statement chain.

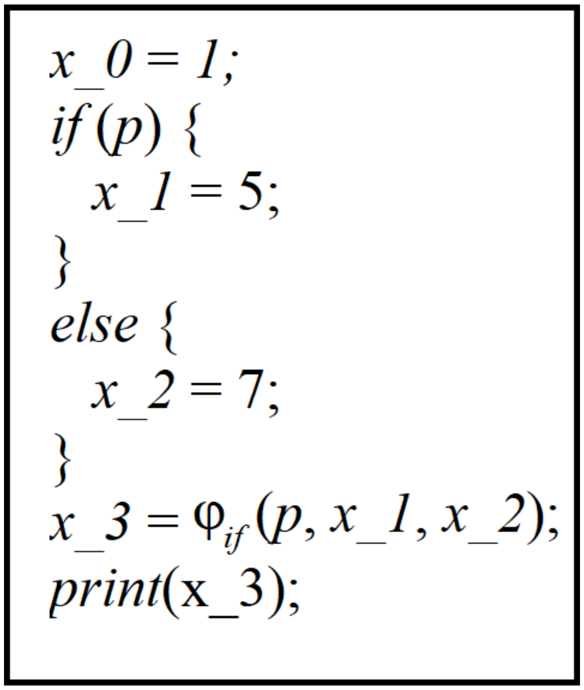
Next, once the modified variables would be known, the actual ϕ_{if} functions would be constructed. As stated in section 2.2, the ϕ_{if} function takes three inputs:

$\phi_{if}(p, v_1, v_2)$. The first input p is the condition to the given *if* statement. Thus, by design, it would be needed that as conditions are approached within the compiler, they must be saved into a list. To be more specific, the conditions that have *already* been modified to account for variable-numbers (i.e. be in the syntax of Figure 3.1) would be saved into a list. After all, if the conditions were saved prior to being translated to GSA form, the variables used within them would not be recognized.

The next two inputs to the ϕ_{if} function are v_1 and v_2, which represent the values of the given variable depending on if the condition p is true or false, respectively. To determine if a given variable were to be changed across an entire *if* statement chain, the hash map method described above could be used; but unfortunately that design does not provide a way to determine when and where each variable was actually assigned. Meaning, currently there would be no way to know whether a given variable was assigned in the *if* statement, one of the *else-if* statements (if there are any), or the *else* statement (if there is one). Thus, it is necessary to have an additional design element to account for where each variable is actually assigned. The design to accomplish this would be a list of hash maps, where each hash map represents a block of the *if* statement (meaning either the *if* statement, one of the *else-if* statements, or the *else* statement). The hash map for each block would contain a mapping from each variable within the block to the *last* variable-number encountered for that variable within the block. Only the last variable-number is needed for each individual block because the last occurrence of each variable is the one that will actually be—potentially—the output of the ϕ_{if} function.

Also, it is important to note that the same design used to track which variables were changed in an entire *if* statement chain—where the variable-number hash map is copied and saved—can not be used for this design as well. This is because the hash maps for each *if* statement block should *only* contain the variables that are actually modified within the given block. The global variable-number hash map would contain the variable-numbers of *all* variables within the Java program, which would provide incorrect information to the design, because the *if* statement block hash maps are designed to represent the last defined variable versions within that given block. Therefore, variables defined outside of the given block should not exist within the hash map for that block.

Once the three inputs would be gathered, the ϕ_{if} functions would be placed at the bottom of the *if* statement chain, and they would be assigned to new variable versions for each of those modified variables. A visual representation of this explanation can be seen in Figure 3.2.

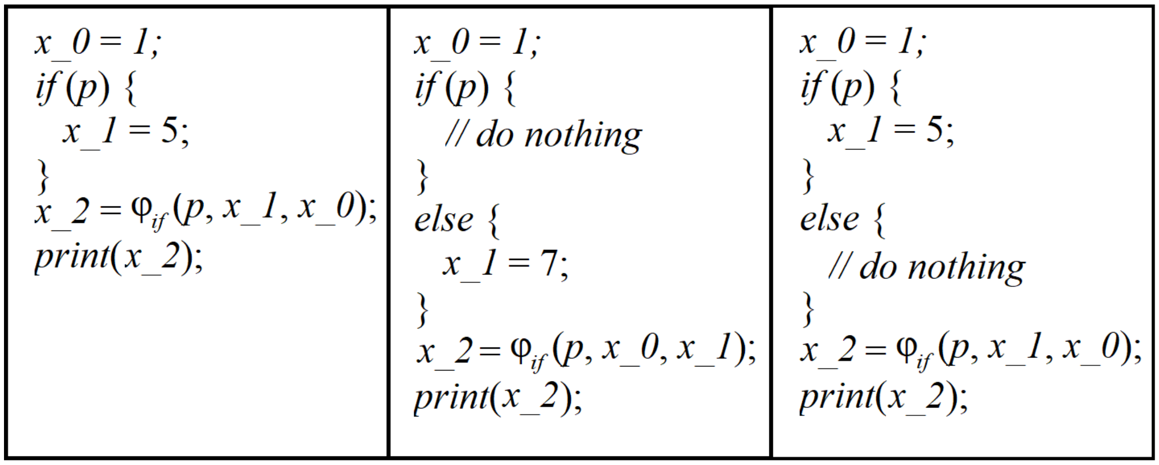


```
x_0 = 1;
if (p) {
  x_1 = 5;
}
else {
  x_2 = 7;
}
x_3 = φ_if(p, x_1, x_2);
print(x_3);
```

Figure 3.2. ϕ_{if} Function Design Example

All of the design aspects explained for *if* statements thus far satisfy the needs for a simple *if-else* statement. However, there are a few *if* statement variants that we need to be able to consider. Imagine an *if* statement that has no *else* statement, or an *if* statement that only modifies a variable in one block of the *if* statement chain (i.e. only in the *if* block or only in the *else* block). Examples of these types of *if* statements can be seen in Figure 3.3. All of these examples require that a few extra conditions are added to the existing design. The first addition is a simple condition that can check whether or not the given *if* statement has an *else* block. If it does not have an *else* block, it is known that if p is false, then the variable version prior to the *if* statement chain should be returned. As for the other examples, additional checks have to added to the existing hash map framework. The design already has a hash map for each block of an *if* statement chain, so when constructing the ϕ_{if} functions, there needs to be a check of whether or not the current variable was changed in the *if* block and whether or not it was changed in the *else* block (by checking each blocks' hash map). If the given variable was *not* changed in one of the blocks, then the definition of the given variable prior to reaching the *if* statement chain would be returned (just how it is displayed in the last two segments of Figure 3.3).

The design for *if* statements has been fully satisfied for *if* statements and *if-else* statements, but what about when we have an *if* statement chain that includes *else-if* statements? To keep consistencies with the existing design, *if* statement chains with *else-if* statements would still be contained in one ϕ_{if} function, but the function would include calls to itself. After all, the ϕ_{if} function uses one condition to decide between two variable versions, so in order to include more decisions,



```
x_0 = 1;
if (p) {
  x_1 = 5;
}
x_2 = φif(p, x_1, x_0);
print(x_2);
```

```
x_0 = 1;
if (p) {
  // do nothing
}
else {
  x_1 = 7;
}
x_2 = φif(p, x_0, x_1);
print(x_2);
```

```
x_0 = 1;
if (p) {
  x_1 = 5;
}
else {
  // do nothing
}
x_2 = φif(p, x_1, x_0);
print(x_2);
```

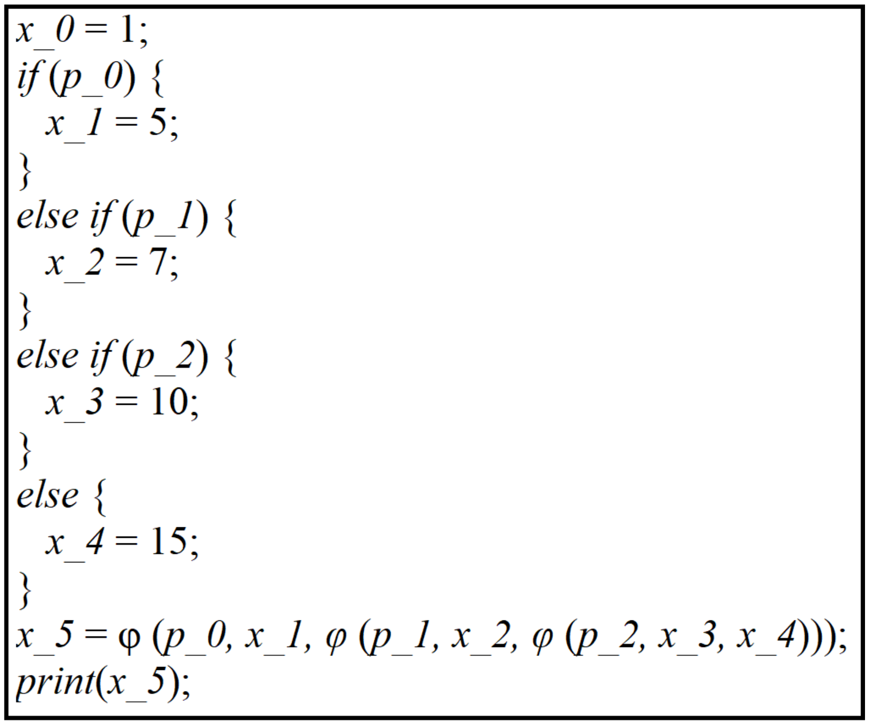
Figure 3.3. Additional ϕ_{if} Function Design Examples

more ϕ_{if} functions would be needed. To be more specific, one extra ϕ_{if} function would be needed for every *else-if* block. This is because every *else-if* chain introduces a new condition, which then has two possible outcomes (true or false). To fully explain this design, Figure 3.4 provides a clear example of an *if* statement chain with multiple *else-if* blocks.

From Figure 3.4, it can be seen that no matter the length of the *if* statement chain, the ϕ_{if} function continues to copy an *if* statement's behavior. When p_0 is true, x_1 is returned, else if p_1 is true, x_2 is returned, else if p_2 is true, x_3 is returned, else, x_4 is returned. Therefore, the ϕ_{if} function still maintains the same semantics as the *if* statement chain it is representing. Plus, it is important to note that all of the design aspects mentioned above would still work with *if* statements of longer lengths, as long as the ϕ_{if} functions are formatted in this way (like in Figure 3.4).

3.1.3 Switch Statements

There is no mention of *switch* statements in GSA literature, but it would be desirable for the GSA compiler to work on as many Java programs as possible. Thus,



```
x_0 = 1;
if (p_0) {
  x_1 = 5;
}
else if (p_1) {
  x_2 = 7;
}
else if (p_2) {
  x_3 = 10;
}
else {
  x_4 = 15;
}
x_5 = φ (p_0, x_1, φ (p_1, x_2, φ (p_2, x_3, x_4)));
print(x_5);
```

Figure 3.4. ϕ_{if} Function Long Chain Example

some improvisation was required. Since *switch* statements behave intensely similarly to *if* statements, the decision was made to translate all *switch* statements into *if* statements, so then they could follow the same GSA conversions as described in the previous section. An example translation is shown in Figure 3.5.

It is important to note the difference between a *switch* statement case where there is not a *break* statement versus when there is a *break* statement. When a *break* statement is not present in a given case, the following case should not be connected to that *if* statement chain, because inside the original *switch* statement, the following case could theoretically be executed as well. However, if a *break* statement is present in a given case, the following case *should* be connected to the *if* statement chain, because inside the original *switch* statement, if that given case were executed it would then break out of the *switch* statement. Meaning, no cases

```
switch(a) {
  case 1:
    // code
  case 2:
    // code
    break;
  default:
    // code
}
```

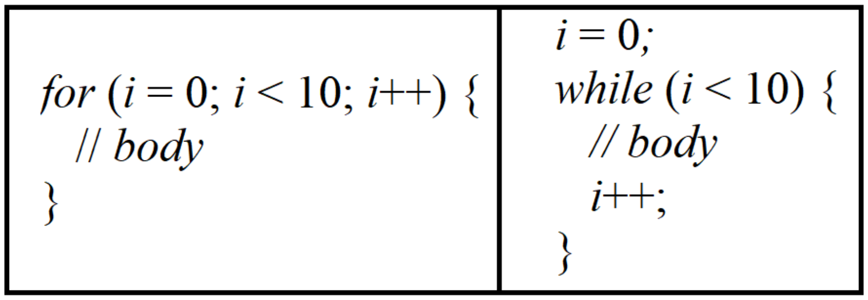
```
if (a = 1) {
  // code
}
if (a = 2) {
  // code
}
else {
  // code
}
```

Figure 3.5. Switch Statement to If Statement Translation

below the given case would be executed as well. This is identical to how *if* statement chains work, because at most one block from the chain will be executed.

3.1.4 Loops

The remaining two gating functions, ϕ_{entry} and ϕ_{exit}, are both related to loops. Java contains three types of loops: *for*, *while*, and *do-while*. Since *for* loops are fairly complex, the design goal would be to translate all *for* loops into *while* loops prior to generating the GSA form of the given Java program. To do so, the initialization step would be moved above the loop, and the iteration step would be moved to the very bottom of the loop's body. Figure 3.6 provides an example of this translation. By transforming all *for* loops into *while* loops, only *while* loops and *do-while* loops need to be considered. However, since *while* loops and *do-while* loops function in nearly identical ways, only generic *while* loops will be discussed for the rest of the design section (for consistency purposes).



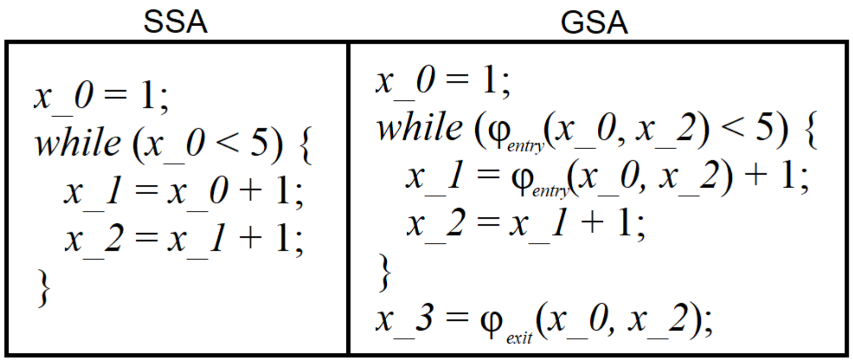
```
for (i = 0; i < 10; i++) {
   // body
}
```

```
i = 0;
while (i < 10) {
   // body
   i++;
}
```

Figure 3.6. For Loop to While Loop Translation

Unlike how the ϕ_{if} function is simply added at the end of an *if* statement chain, the ϕ_{entry} function needs to replace existing code. Specifically, the ϕ_{entry} function needs to replace every reference to a variable in a loop given that the variable has not been assigned within the loop itself. This includes all variables used within the loop's condition as well. Figure 3.7 displays how the ϕ_{entry} functions should be placed. It is important to note that x_1 is not replaced with a ϕ_{entry} function on line 4 because x_1 has already been assigned to a ϕ_{entry} function in the previous line. Also, just to reiterate, the ϕ_{entry} function is meant to determine between the initial variable and the most recently iterated version of the variable in the loop. Thus, that is why the function is choosing between x_0 and x_2, because x_0 is x's definition prior to the loop, and x_2 is the last definition of x within the loop's body. The variable-number hash maps before and after the loop's execution would be used to determine these values.

Knowing where to place the ϕ_{entry} functions is simple, but in order to fill them with the correct information, future details are needed. For example, when reaching the condition of a *while* loop, it is known that all variables used within it would need to be replaced by a ϕ_{entry} function, so both the initial definitions of those variables and the last iterated definitions of those variables within the loop would



SSA

```
x_0 = 1;
while (x_0 < 5) {
    x_1 = x_0 + 1;
    x_2 = x_1 + 1;
}
```

GSA

```
x_0 = 1;
while (φentry(x_0, x_2) < 5) {
    x_1 = φentry(x_0, x_2) + 1;
    x_2 = x_1 + 1;
}
x_3 = φexit(x_0, x_2);
```

Figure 3.7. While Loop in SSA Form vs. GSA Form

need to be known. The initial versions of those variables could be obtained easily, because they would be obtained from calling on the variable-number hash map at that point in the code. However, how would the last iterated versions of those variables within the loop be obtained? After all, the loop's body would not have been reached at this point. Therefore, rather than replacing variables with ϕ_{entry} functions as they are encountered, the design is to just save their locations as they are encountered. Once the *while* loop is traversed entirely, all of those locations can be revisited and replaced with ϕ_{entry} functions; because at that point, the last defined versions of each variable within the loop would be known, so all of the necessary information would be known to populate the ϕ_{entry} functions.

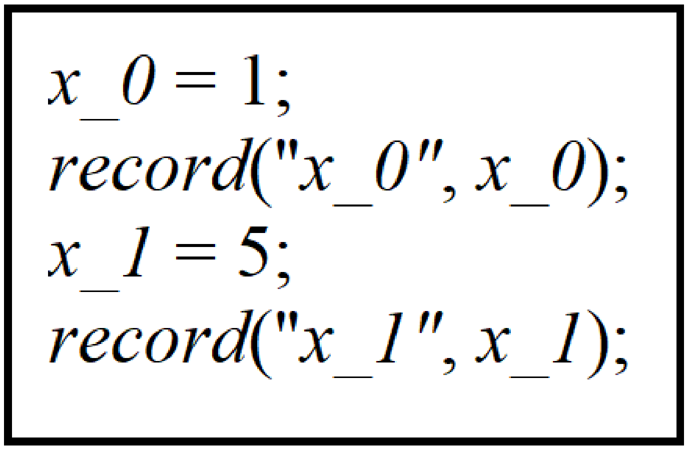
As for the ϕ_{exit} function, the design is similar to the ϕ_{if} function. After all, for every variable that is modified within a given *while* loop, there must be a ϕ_{exit} function placed at the end of the *while* loop for it. Since the ϕ_{exit} function must decide between the definition of the given variable prior to the loop and the last version of the variable within the loop, it can utilize the same version numbers obtained for the ϕ_{entry} functions. An example of a ϕ_{exit} function can be seen in Figure 3.7.

The ϕ_{if} function decides which variable version to return based off of the condition input p. However, the ϕ_{entry} and ϕ_{exit} functions do not have a condition input, so there needs to be a way to determine which variable version to return. To do so, both functions will check to see if the v_{exit} has been defined. If the v_{exit} has been defined, it will be returned, if not, v_{init} will be returned (which is the variable version prior to the loop). In Java, this design is generally not plausible, because variables that have not been defined cannot be inputs to functions. However, in the implementation section of this chapter it will be explained how this design was successfully implemented.

3.1.5 Data Recording

For value-based statistical fault localization, the value of each variable needs to be known at the end of each test case. This implies the necessity for a way to record variable values during program execution. The design to accomplish this would be to add "record" statements after every variable assignment which would output all variable-value pairs into a text file. The record statements would need to be included in the design, and they would allow for variable values to be known after each test execution. Figure 3.8 showcases what these record statements would look like. For the purpose of this thesis, only the variable name and variable value are necessary for recording.

Value-based statistical fault localization also requires each test execution to result in either a "pass" or a "fail" (or in causal inference terminology, an outcome variable Y with value 0 or 1). However, this requirement brings forth multiple limitations to our design:



```
x_0 = 1;
record("x_0", x_0);
x_1 = 5;
record("x_1", x_1);
```

Figure 3.8. Variable Value Record Statements

- First of all, in order to determine if a test case "passed" or "failed", it is necessary that the correct output is known. Therefore, if there exists a faulty program with an unknown correct output, our design would not be able to correctly rank the suspiciousness scores of all of the variables in an empirical study.
- In many cases a program will not have a single output. Also, there are many cases where a developer may be more interested in checking an outcome that is not the final output of the program. Meaning, in the eyes of the compiler, it is unclear where the "output" should be recorded.

The limitations above require the necessity for developer intervention. The ideal design would be entirely automated, but there is no current way to bypass the limitations listed above. Thus, under the current design, developers must have a faulty and non-faulty version of the program of interest. Or, at the very least, the developers must be aware of what the desired output should be, even if they do not have a non-faulty iteration of the program of interest. Also, the second limitation requires the developer to manually insert where to record the program's "output". To do so, the design includes a "record-output" function, which should be placed wherever the desired output is returned within the program of interest. This function will print the output to the same text file that all variable-value pairs are printed to. However, the "record-output" function will label the output differently, so it can be singled out in an empirical study.

Although the loss of an entirely automated process is upsetting, by allowing the developer to place the "record-output" function anywhere they desire, it actually provides the design with a lot of flexibility. After all, it permits the developer to test any section of their program that they may suspect has a fault. With an entirely automated process, the developer would not have this type of freedom.

3.1.6 Causal Map

A causal map is a causal inference data structure that represents the dependency relationships of the objects of interest [7]. In this thesis, the causal map would need to consist of the GSA variables within the program of interest, and map each variable to all of the variables that were included within that variable's assignment statement. Since the compiler transforms the program into GSA form, all variables are assigned once, so we know for certain all of the variables that a given variable depends on.

To actually create the causal map, a hash map will be used that maps variables to lists of variables. In the compilation process, whenever a variable is assigned, it will be added as a key to the causal hash map, then all variables included in the right-hand side of the assignment expression will be added to a list and mapped to the variable key. For example, if the following assignment statement is encountered:

$$x_2 = x_1 + y_0;$$

then the following key-value pair will be added to the causal hash map: $(x_2, (x_1, y_0))$. This represents the meaning that the variable x_2 is dependent on the variables x_1 and y_0.

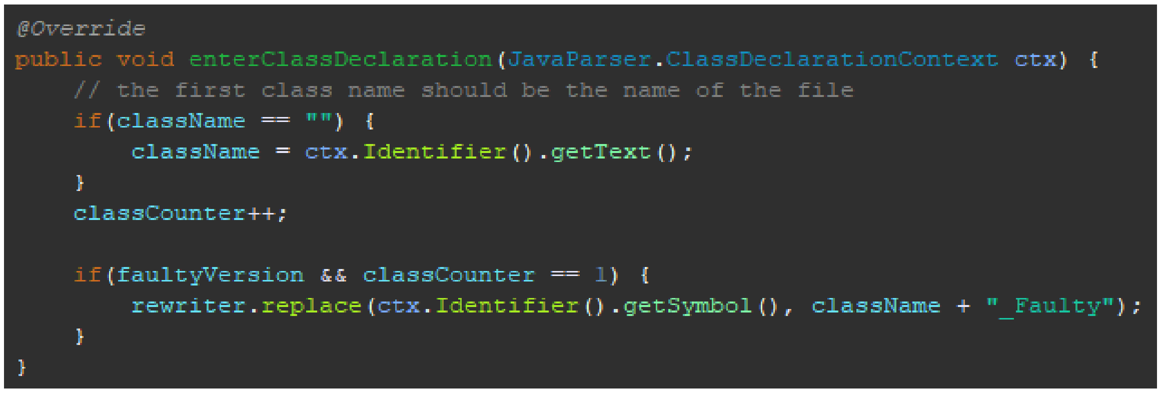
Once the entire program of interest is traversed and all variables have been added to the causal hash map, the hash map then needs to be saved externally. To do so, the design would simply loop through the hash map and print every key-value pair to a text file. This allows the developer to have access to the causal dependencies for all variables within the GSA Java program. This is crucial because a causal map is needed to implement the causal inference techniques into the statistical fault localization process. Without a causal map, confounding bias would not be dealt with.

3.2 Implementation

3.2.1 ANTLR4

In order to create the GSA compiler, a way to parse through Java files was necessary. ANTLR (ANother Tool for Language Recognition) was used for this purpose [3]. The tool is extremely flexible, as it can take any grammar and produce a parser for it that can be used to read, process, execute, or translate files that obey the given grammar [3]. To be more specific, ANTLR4 uses a given grammar file to produce a variety of files: parsers, lexers, listeners, and visitors. When a program—that obeys the given grammar—is run through ANTLR4, a parse-tree for that program is created using the ANTLR4-generated files [3]. ANTLR4 also generates "tree walkers", which give developers the freedom of visiting any nodes within the parse-tree for the purpose of analyzation or modification [3].

For this thesis, an existing Java grammar file [8] was used for ANTLR4. Once Java files are inputted into the framework, the primary ANTLR4 feature that is utilized is the parse-tree walker. Using this ANTLR4 feature, every parse-tree node can be visited and modified at both entrance and exit. Meaning that as the parse-tree is traversed—in a preorder traversal scheme—whenever a node is entered, it can be visited and modified, and whenever that same node is exited later, it can be visited and modified once again. This flexibility permits functionality to differ between when a node is being entered or exited, which can be quite useful. All of these "enter" and "exit" functions take a special *context* variable as input, which includes all of the grammar-specific elements that are present at that point in the parse-tree. Figure 3.9 showcases one of the "enter" methods that can be overridden. The ctx input has type $JavaParser.ClassDeclarationContext$, which includes all of the necessary contextual information that should be known at a Java class declaration. Within the code, the $Identifier()$ field is used to return the given Java file's class name. This is a simplistic example of how ANTLR4 visitor functions can be used to obtain information from the code being parsed.



```
@Override
public void enterClassDeclaration(JavaParser.ClassDeclarationContext ctx) {
    // the first class name should be the name of the file
    if(className == "") {
        className = ctx.Identifier().getText();
    }
    classCounter++;

    if(faultyVersion && classCounter == 1) {
        rewriter.replace(ctx.Identifier().getSymbol(), className + "_Faulty");
    }
}
```

Figure 3.9. *enterClassDeclaration* Method

Figure 3.9 also showcases the use of the *rewriter* variable. The *rewriter* variable is of type *TokenStreamRewriter*, which is an object type from ANTLR4. *TokenStreamRewriter* objects can be used to modify the code that is currently being parsed. Thus, the *rewriter* object is what is being used to actually translate the input Java program into the output Java program (which will be in GSA form). Fortunately, the *rewriter* object has tons of flexibility, as it allows for text to be appended after a token, before a token, or replace the token entirely. This is largely why ANTLR4 was chosen to implement source-to-source translation for this thesis.

3.2.2 Pre-Processing Translation Pass

As discussed in the design section, there are a few Java statement types that need to be translated to other statement types. The purpose of doing these translations was so multiple GSA translations would not be necessary. For example, the design states that all *for* loops need to be translated to *while* loops prior to translating them to GSA form. Thus, only a GSA translation for *while* loops will be necessary.

In order for these translations to be made prior to any GSA conversions, a pre-processing parsing pass needs to done on the input Java file, which will only focus on these translations. Along with the *switch* statement and *for* loop translations, there are a few other translations that were decided on to simplify the GSA transformation process. All of the primary focuses of the pre-processing translation pass are listed below:

- Transform all *switch* statements into *if* statements.
- Transform all *for* loops into *while* loops.
- Transform all types of assignment statements into generic assignment statements. Examples: change *x += 1* into *x = x + 1*, change *x *= y + 5* into *x = x * (y + 5)*, etc.
- Save all global variables into a list.

The reasoning for translating all types of assignment statements into generic assignment statements is the same as why all *for* loops are being translated to *while* loops. It simply makes the GSA conversion process much easier, as it guarantees all assignment statements will be of the same form, so only one GSA transformation is necessary for assignments. Plus, GSA form requires that all assignment statements are generic assignment statements, because in GSA form, every assignment statement is also a variable declaration. So by making the translation beforehand, it makes the GSA translation code less cluttered.

As for the global variable list, this is necessary because—in Java—global variables can be referenced on a line number that is lower than the line number of where the global variable is actually declared. On the other hand, in ANTLR4, when the parse-tree is being walked, it will traverse the code line-by-line. This means that if a global variable reference is located prior to the global variable's declaration, there will be no way to determine what type of variable has been encountered. This issue is only important for the GSA transformation pass, because it needs to be known that a global variable has been encountered, so it can be ignored (since global variables are not included in the scope of this thesis). Thus, the pre-processing translation pass saves all global variables, and will pass the list to the GSA transformation pass so it will always know which variables are global variables, so they can be handled correctly.

3.2.3 GSA Transformation Pass

The first implementation focused on for the GSA transformation pass was translating all variables into a new object type. The reasoning for this implementation is because of scoping issues in Java. For example, look back at Figure 3.2, and it can be seen that the ϕ_{if} function takes *x_1* and *x_2* as inputs. However, *x_1* and *x_2* were declared within the *if* and *else* statement blocks. Meaning, in Java, those variables would not be in-scope where the ϕ_{if} function is called on. Therefore, the design needed a way for all variables to be in-scope, even if they have not been assigned a value yet. This was accomplished by creating a new object type called "Var", which contains a field called "value". Using this framework, all variables can be assigned to *null* at the top of the method they are declared in. Thus, even if they have not been declared an actual value, they can still be referenced in phi functions.

By creating an object type for all variables, it also simplifies the implementation of phi functions. After all, an undeclared variable will simply be null. The implementation for the 3 types of gating functions are described below:

- $\phi_{if}(p, v_1, v_2)$: this implementation is identical to the theory, as it simply returns v_1 if p is true, and it returns v_2 if p is false.
- $\phi_{entry}(v_{init}, v_{iter})$: by using our "Var" object type, undeclared variables will be null. Thus, the implementation checks if v_{iter} is null or not. If it is null, v_{init} is returned. Otherwise, v_{iter} is returned, because it being non-null implies that it has been declared.
- $\phi_{exit}(v_{init}, v_{exit})$: this implementation is identical to the ϕ_{entry} implementation. If v_{exit} is null, v_{init} is returned. Otherwise, v_{exit} is returned.

The rest of the design elements were implemented using the ANTLR4 visitor functions, which Table 3.2 goes over in detail:

Method Name	**Method Description**
$enterClassDeclaration$	Obtains the class name for the purpose of creating all of the output files (which will use the class name in their titles). Also, appends "_Faulty" to the end of the class name when creating the faulty iteration of the program.
$enterInterfaceDeclaration$	Obtains the interface name for the purpose of creating all of the output files (which will use the interface name in their titles). Also, appends "_Faulty" to the end of the interface name when creating the faulty iteration of the program.

$enterConstructorDeclaration$	Obtains constructor name, to be included in record statements for variables declared within it. Also, appends "_Faulty" to the end of the constructor identifier when creating the faulty iteration of the program.
$exitConstructorDeclaration$	Informs the code that the constructor has been left, so "Var" null declarations will no longer be made in the constructor.
$enterConstructorBody$	Informs the code that the first line of the constructor needs to be located for "Var" null declarations.
$exitConstructorBody$	Used to ensure that constructors start with *super* or *this* calls.
$enterMethodDeclaration$	Gets current method name for recording purposes.
$exitMethodDeclaration$	Erases current method name.
$enterFormalParameters$	Creates a hash map to contain all of the formal parameters for this current method.
$enterFormalParameter$	Adds this formal parameter and it's type to the formal parameter hash map for this current method.

$enterMethodBody$	Informs the code that the first line of the method needs to be located for "Var" null declarations.
$exitMethodBody$	Informs the code that the method has been left, so "Var" null declarations will no longer be made in the method.
$enterBlock$	Adds a layer to the program's scope. Also, checks if this is the entrance to a method or constructor, and if it is, sets the first line (which is where "Var" null declarations will be made).
$exitBlock$	Pops a layer from the program's scope.
$enterBlockStatement$	Inserts a comment to separate the null declaration section from the actual code section.
$enterStatement$	Checks to see if the statement is an *if* statement or a *while* loop. From there, it will initialize all of the required data structures described in the design section for that given statement.
$exitStatement$	Checks to see if the statement is an *if* statement or a *while* loop. From there, it will utilize the corresponding data structures to insert all of the phi functions for that given statement.

$enterLocalVariableDeclaration$	Obtains variable type, and deletes the type (i.e. transforms $int\ x = 0$ into $x = 0$).
$exitLocalVariableDeclaration$	Initializes the "Var" object, and inserts the record statement for this variable.
$enterVariableDeclarator$	Increases the variable count for this variable, adds a causal map entry for it, and inserts the null declaration at the top of the current method or constructor.
$exitVariableDeclarator$	Informs the code that this declaration has been left.
$enterExpression$	Checks if this expression is an assignment expression. If it is, the assigned variable has it's variable count incremented, a causal map entry is added for it, and the null declaration is added at the top of the current method or constructor.
$exitExpression$	Checks if this expression is an assignment expression. If it is, the assigned variable is initialized as a "Var" object, and a record statement is inserted for the assigned variable.
$enterParExpression$	Manages the depth of if conditions and while conditions.

$exitParExpression$	Manages the depth of if conditions and while conditions. If this is the final closing parenthesis, the code is informed that the condition has been left.
$enterPrimary$	Manages all variable references. Dependent on where the variable is being visited, the correct variable version needs to be appended. This is done through a long list of conditionals.
$enterCreatedName$	In the faulty iteration of the program, "_Faulty" is appended to class objects.

Table 3.2. GSA Transformation Pass Visitor Methods

The above methods work together to implement all of the design aspects discussed in section 3.1. Thus, with the pre-processing pass and this GSA transformation pass, a Java input file can be successfully translated into GSA form (alongside a corresponding causal map file).

3.2.4 Statistical Fault Localization

The design elements implemented in the sections above provide a way of (1) translating a Java program into an equivalent Java program in GSA form, (2) recording variable-value pairs during run-time, (3) recording program output values during run-time, and (4) creating a causal map of all the variables in the program. All of these design elements need to be combined to actually execute statistical fault localization. However, before that is possible, there are a few other files that need to be added to the output of the GSA compiler:

- A main Java program that will execute all of the test cases on both the faulty and non-faulty Java files, and then compare the outputs.
- An R file that will create a table of all GSA variables and their corresponding confounding variables (i.e. all of the variables from the causal map).
- An R file containing the implementation of CounterFault, which will be used to produce the suspiciousness score rankings.

By default, the main Java program runs the main method on both the faulty and non-faulty versions of the program of interest. However, as discussed in section 3.1.5, the developer has the final decision of what output is actually being tested. Implying that if the output being tested does not belong to the program's main method—or if the program does not have a main method—the developer will have to modify the code to call on the correct method.

The R file with the table of all the confounding adjustment variables will be generated by the main Java program. This happens by iterating through the causal map text file, obtaining all the key-value pairs, and transferring them into an R table, using R's formatting.

The CounterFault R file is provided in the project directory. Thus, it is simply copied into the output directory for the current program being tested. In this thesis, no adjustments were made to this file. The same R file from [14] is utilized. Once the main Java program has been executed, and all of the test cases—for both the faulty and non-faulty Java files—have been run, this R file should be executed. It will produce a CSV file, which will rank every variable from their suspiciousness scores in decreasing order. Therefore, providing the developer with the first statements to check for flaws.

These files will be used to test the tool in the section to follow, through an in-depth empirical study.

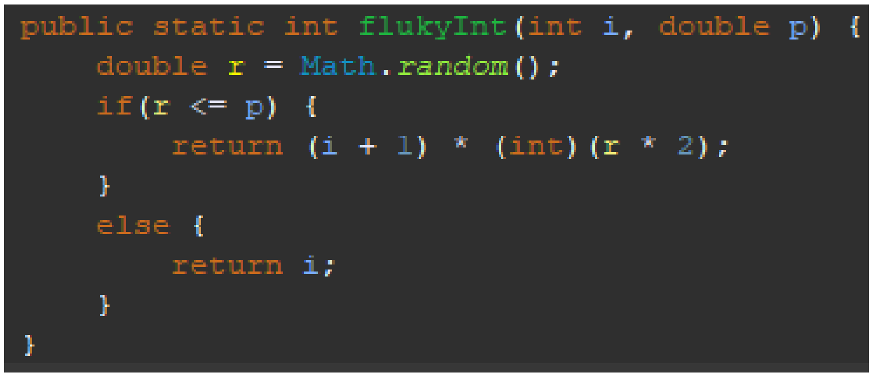
4 Empirical Study

4.1 Study Design

In order to evaluate how this tool compares to the tools developed in Sheng [20] and Roach's [19] studies, it was decided that an in-depth empirical analysis would be done on a variety of numerical Java programs. Specifically the same set of 10 numerical Java programs that were tested on in [20] and [19]. The 10 programs each come from one of the following Java libraries:

- Apache Commons Math 3.6.1 [2]: simplistic mathematics and statistics library that includes common math functions that are not available in Java by default.
- JAMA [11]: generic linear algebra library for Java.
- SciMark 2.0 [1]: Java library used for scientific and numerical computation.

In order to evaluate this tool, each test program needs to have faulty versions. Thus, a total of 3 faulty versions were constructed for each test program. This was done by inserting fault-inducing functions into 3 random assignment-locations of each test program (random assignment-locations were chosen to avoid selection bias). These fault-inducing functions are contained within a class called "Fluky", which contains fault-inducing functions for a variety of numerical variable types. Figure 4.1 showcases an example fluky function for integer variables. The function takes the correct integer value i as input, and it also takes a double p as input, which represents the probability of a fault occurring. For example, if p is equal to 50%, there is a 50% chance that the correct value will be returned, but there is also a 50% chance that $(i+1)*(int)(r*2)$ will be returned. When the latter occurs, that variable assignment will officially be turned into a fault.



```
public static int flukyInt(int i, double p) {
    double r = Math.random();
    if(r <= p) {
        return (i + 1) * (int)(r * 2);
    }
    else {
        return i;
    }
}
```

Figure 4.1. Fault-Inducing Function for Integers

Each test program has 3 faulty versions to test different types of faults in different types of locations. Along with fault types and locations, the frequency of faults also needs to be tested. For that reason, each faulty version of each program will be tested at 4 different probabilities of fault-occurrence: 25%, 50%, 75%, and 99%. This means that in total each test program will have 12 variations. For each one of those variations, the test program will be executed 1000 times. For each of those executions, the variable assignments and desired outputs will be recorded to the "output" text file. The variable assignments from all 1000 executions will be used to form a table in a new text file called "newoutput". The table will contain the variable-value pairs for each execution of the test program, and it will be used for the statistical fault localization analysis.

Program	**# of Lines**	**# of Executions**	**Faulty Versions**
Apache.FastCosineTransformer	183	1000	3
Apache.FastSineTransformer	182	1000	3
Apache.LUDecomposition	398	1000	3
Apache.SplineInterpolator	130	1000	3
Jama.CholeskyDecomposition	220	1000	3
Jama.LUDecomposition	344	1000	3
Jama.QRDecomposition	247	1000	3
Jama.SingularValueDecomposition	592	1000	3
SciMark.FFT	202	1000	3
SciMark.LU	286	1000	3

Table 4.1. Test Programs

As for the recorded outputs, they will be used to determine if each execution passed (0) or failed (1). To do so, the non-faulty version of each test program will be executed, and the recorded outputs—of the faulty version of the program—will be compared to the outcome of the corresponding non-faulty program. Once an execution has been determined to pass (0) or fail (1), a 0 or 1 will be appended to an "outY" file within the current program's output directory. Thus, after all 1000 executions for a given program version, there will be a completely populated "outY" text file, which will contain the result of each execution. This will be used for the statistical fault localization calculations.

Once all 1000 executions have been run for a given test program version, the last step is running the statistical fault localization analysis. Like mentioned earlier in this thesis, the CounterFault framework is being used, so the CounterFault R file simply needs to be run to obtain all of the suspiciousness scores for the current test program version.

To make things as clear as possible, the process will be walked through for a single test program version:

- First, run the program through the GSA source-to-source compiler. This will result in the following output files:
  (1) Non-faulty Java program in GSA form.
  (2) Faulty Java program in GSA form.
  (3) Causal map text file which contains all causal relationships.
  (4) Main Java program used for running the non-faulty and faulty Java files on the desired methods and with the desired inputs.
  (5) An R file that creates a table of all GSA variables and their corresponding confounding variables.
  (6) An R file containing the implementation of CounterFault (for statistical fault localization)
- Next, the developer will modify the main Java program to call on the desired methods to be tested, and to adjust the inputs.
- The developer will also need to manually add the "record-output" statements within the faulty and non-faulty Java files at the desired locations.
- The developer will also need to manually insert the faults into the faulty Java program using the "Fluky" class.
- Moving on, the main Java program will be run, which will go through all 1000 executions for the faulty program. This will result in the following output files:

(1) "newoutput.txt": a table with the variable values for each of the 1000 executions. If a variable was not reached for a given execution, it is given the value: "NA".

(2) "outY.txt": an array of 0s and 1s, where each number corresponds to one of the 1000 executions. 0s represent executions that passed, and 1s represent executions that failed.

- Finally, the CounterFault R file will be executed in an R environment. It utilizes the R file with the confounding adjustment variables, the "newoutput.txt" file, and the "outY.txt" file for it's fault localization calculations. This will result in a CSV file that contains the suspiciousness scores for all variables within the test program version, ranked in decreasing order.

The entire process above was completed for all 12 versions of each test program. The results will be discussed in the following section.

4.2 Results and Analysis

Tables 4.2, 4.3, and 4.4 display the rankings of the faulty variables for each test program version in terms of suspiciousness score in decreasing order. A ranking of 1 implies that the given variable is the most suspicious variable in the program, according to *CounterFault*.

In this empirical study, 69 of the 120 faulty programs labeled the faulty variable as the most suspicious variable over all. Meaning that the majority of the time—57.5% of the time, to be exact—this tool was completely correct. Furthermore, 109

	***p* = 25%**	***p* = 50%**	***p* = 75%**	***p* = 99%**
Apache.FastCosineTransformer	1/36	1/36	1/36	36/36
Apache.FastSineTransformer	1/27	1/27	1/27	27/27
Apache.LUDecomposition	1/236	1/236	1/236	169/236
Apache.SplineInterpolator	1/28	1/28	1/28	1/28
Jama.CholeskyDecomposition	1/72	1/72	1/72	2/72
Jama.LUDecomposition	1/167	1/167	1/167	167/167
Jama.QRDecomposition	6/185	5/185	8/185	5/185
Jama.SingularValueDecomposition	1/517	1/517	1/517	1/517
SciMark.FFT	1/135	1/135	1/135	128/135
SciMark.LU	1/190	1/190	1/190	187/190

Table 4.2. Program Fault Version 1: Faulty Variable Ranking

	***p* = 25%**	***p* = 50%**	***p* = 75%**	***p* = 99%**
Apache.FastCosineTransformer	1/36	1/36	1/36	36/36
Apache.FastSineTransformer	1/27	1/27	1/27	2/27
Apache.LUDecomposition	1/236	1/236	1/236	236/236
Apache.SplineInterpolator	1/28	1/28	1/28	1/28
Jama.CholeskyDecomposition	1/72	1/72	72/72	72/72
Jama.LUDecomposition	5/167	9/167	7/167	1/167
Jama.QRDecomposition	3/185	3/185	3/185	185/185
Jama.SingularValueDecomposition	10/517	5/517	3/517	1/517
SciMark.FFT	2/135	2/135	3/135	3/135
SciMark.LU	8/190	13/190	13/190	1/190

Table 4.3. Program Fault Version 2: Faulty Variable Ranking

of the 120 programs (90.8%) labeled the faulty variable in the top 7% of the suspiciousness score rankings. Therefore, when comparing the tool on its own, it is mostly effective, and thus shows that this methodology could be highly useful for the software debugging process.

However, now the drawbacks of the tool will be addressed. First of all, the tool does not function well with faults that occur very often. Although this may seem odd—because it would be expected that the more a fault occurs, the more likely it

	***p* = 25%**	***p* = 50%**	***p* = 75%**	***p* = 99%**
Apache.FastCosineTransformer	1/36	1/36	1/36	17/36
Apache.FastSineTransformer	1/27	1/27	1/27	16/27
Apache.LUDecomposition	1/236	1/236	2/236	2/236
Apache.SplineInterpolator	1/28	1/28	1/28	6/28
Jama.CholeskyDecomposition	3/72	4/72	5/72	1/72
Jama.LUDecomposition	5/167	5/167	8/167	1/167
Jama.QRDecomposition	5/185	4/185	4/185	1/185
Jama.SingularValueDecomposition	1/517	1/517	1/517	271/517
SciMark.FFT	1/135	1/135	1/135	1/135
SciMark.LU	2/190	2/190	2/190	1/190

Table 4.4. Program Fault Version 3: Faulty Variable Ranking

could be pointed to a cause—this actually lines up with how *CounterFault* functions. After all, *CounterFault* works the best when faced with "coincidental correctness" [18]; which is when a faulty variable only results in a fault a fraction of the time [15]. This can be logically inferred by understanding how *CounterFault* examines variables and their corresponding values. For example, consider a variable A that can sometimes take value $A = 1$ and sometimes take value $A = 0$, and these values result in the program either failing ($Y = 1$) or not failing ($Y = 0$), respectively. Now imagine a large number of test executions are run, and a table of variable values are compiled for each test execution. If there exist a decent number of cases where $A = 1$ and a decent number of cases where $A = 0$, *CounterFault* will more easily be able to detect a pattern between $A = 1$ and $Y = 1$. Whereas if in every single test execution A were to equal 1, there would be a fault every time, and there would be no alternate cases to compare them to. Thus, *CounterFault* would be unsure which variable is causing the fault, because it does not know how the variables behave when there is no fault present. This issue caused the tool to have significant trouble in ranking the faulty variable correctly in many of the cases where the fault probability was 99%. All of the cases where this issue was prevalent—meaning the ranking was not in the top 7%—are highlighted in pink on Tables 4.2, 4.3, and 4.4. Not only are the results not in the top 7%, most of them are significantly wrong. After all, more than half of the highlighted cases ranked the faulty variable as the *least* suspicious variable. This implies that *CounterFault* has a significant dependence on coincidental correctness, and therefore needs a solution to work around this in the future. Although, in the real world, extreme cases (i.e. where a fault is very likely or very unlikely) tend to be rare.

It is important to note why the *Jama.CholeskyDecomposition* version 2 file resulted in a horrible ranking for the 75% case. The variable that was randomly selected for a fault insertion was an iterative variable located in a double-nested loop. Meaning, the assignment statement had a high probability of being reached multiple times in each test execution. Thus—even though the probability of the fault occurring was only 75%—because it would be encountered so many times per test execution, the probability of a fault occurring was highly increased. The "outY" file was examined, and there were only 2 cases where no fault occurred. So, just like many of the 99% cases, *CounterFault* had little information to work with, and the suspiciousness score derivations were highly effected.

While all of the cases that were not affected by a lack of coincidental correctness ranked the faulty variable in the top 7%, there were some test program versions that performed slightly worse than others:

- *Jama.QRDecomposition* version 1
- *Jama.LUDecomposition* version 2
- *Jama.QRDecomposition* version 2

- *Jama.SingularValueDecomposition* version 2
- *SciMark.LU* version 2
- *Jama.CholeskyDecomposition* version 3
- *Jama.LUDecomposition* version 3
- *Jama.QRDecomposition* version 3

The reason for the slight decrease in performance in these files is believed to be due to array and object usage. Meaning, variable assignments that depend on arrays or object variables. The current tool does not deal with arrays or objects, only generic numeric variables. Thus, variables that rely on array elements or objects cannot adjust for those variables when trying to adjust for confounding bias. The test program versions clearly reflect this issue, because all but one of them are from the JAMA library [11], which utilizes a large amount of arrays and matrix objects. The solution to this issue would be to implement array SSA form, which can be extended for objects as well [17]. This concept will be discussed in further detail in Chapter 6.

Over all, when individually analyzing the effectiveness of this tool, the outcome is generally positive. 90% of the test program versions rank the faulty variable in top 7% of suspicious variables, which makes the debugging process significantly easier for a developer. Although the tool often fails when dealing with faults that occur almost always, these cases are rare in reality. Also, although the tool tends to perform less well when dealing with arrays and objects, this issue can be fixed relatively easily in a future iteration of this tool by adding array SSA form to the source-to-source compiler's functionality.

4.3 Comparison to Previous Studies

Since Roach's study [19] was the most recent iteration of a comparable tool, that will be the primary focus for comparison. Plus, 9 of the 10 test programs analyzed in the empirical study from [19] were used in this thesis's empirical study. This provides a comparable data set, and eliminates any forms of bias around the selection of programs. Furthermore, Roach's study also provides tables that clearly rank the faulty variables in the same way that they are ranked in this study, providing a clear way to compare results. Lastly, Roach's study also utilized *CounterFault* for the fault localization rankings, so the only difference between this thesis's tool and Roach's tool, are the GSA source-to-source compilation tools.

In [19], only 17 of the 116 programs (14.7%) ranked the faulty variable as the most suspicious variable; whereas the tool in this thesis ranked the faulty variable as the most suspicious in 69 of the 120 programs (57.5%). Also, in [19], only 41 of the 116 programs (35.3%) ranked the faulty variable in the top 7% of suspicious variables; whereas the tool in this thesis ranked the faulty variable in the top 7% of suspicious variables in 109 of the 120 programs (90.8%). Clearly, the tool from this thesis greatly outperformed the tool from [19] in terms of correctness. Plus, not only did this tool outperform [19]'s tool, but [19]'s empirical study did not even take measurements at high fault probabilities; which is where the tool from this thesis received most of it's flawed results.

Although the tool from this thesis proportionally outperformed the tool from [19], the results should be considered in a different way as well. Since this thesis implemented pure GSA form, each translated Java program ended up with far more GSA variables than Roach's study. For example, in Roach's study, the converted version of *SciMark.FFT* had a total of 68 GSA variables [19]. On the other hand, in this thesis the converted version of *SciMark.FFT* had a total of 135 GSA variables; which is more than double the amount of GSA variables. Due to this—for Roach's study—ranking the faulty variable in the top 7% of suspicious variables can be fairly difficult in some cases. For example, consider the *Jama.LUDecomposition* program. In Roach's study, there was a case that ranked the faulty variable as the 4th most suspicious variable out of 41 total variables. Whereas in this study, there was a case where the faulty variable was ranked as the 8th most suspicious variable out of 167 variables. The fraction 8/167 is much smaller than the fraction 4/41, but in a real debugging process, a developer would probably prefer to only look through 4 variables instead of 8. Nonetheless, this example just explains why there were so few faulty variables that were ranked in the top 7% of suspicious variables for Roach's study. But even when ignoring proportional results, the tool from this thesis still outperformed Roach's tool over all. After all, even with significantly more GSA variables, this tool was able to locate the faulty variable as the most suspicious variable 57.5% of the time versus Roach's tool which could only do it 14.7% of the time.

In [19], Roach mentions that Sheng's tool [20] resulted in nearly identical results, but Sheng's tool did outperform Roach's tool in cases that relied on arrays. After all, Sheng's tool did implement an inspired version of array SSA form [20]. Therefore, Sheng's tool was able to perform well for programs that utilized arrays; which is something that this tool does not include. Nonetheless, even without implementing a way for this tool to handle arrays or objects, the results were only slightly negatively impacted for those cases.

4.4 Threats to Validity

By closely replicating existing empirical studies [19, 20] clear comparisons could be made between previous tools and the tool from this thesis. However, performing well in this empirical study does not guarantee the effectiveness of this tool over all. After all, the set of numerical Java programs in this study is not very extensive. The largest file is only 592 lines of code, which is pretty small when considering large software systems. Therefore, this empirical study can only ensure the tool's functionality in small numerical programs. As for large Java programs, the results from this empirical study can only estimate that the tool would perform well in those cases as well.

Furthermore, just to reiterate, this tool only has functionality for numerical Java variable types (int, long, short, byte, float, and double). There is currently no functionality for any other types of variables, like arrays, strings, objects, etc (i.e. non-primitive types). Although numerical Java programs were handpicked to avoid non-primitive types, it is virtually impossible to find Java programs with decent complexity that do not contain at least some non-primitive types. Therefore, the presence of non-primitive types within some of the test programs is a threat to the validity of this study, because they are unaccounted for in the statistical fault localization calculations.

Lastly, it was shown that this tool's validity is highly dependent on the coincidental correctness of a fault. Meaning, in order for *CounterFault* to correctly rank the faulty variable as a highly suspicious variable, there needs to exist a decent proportion of cases where the faulty variable does not result in a fault. In cases where faults occur in nearly every test execution, *CounterFault* has trouble locating patterns between variable values and program faults, and thus the validity of this tool is challenged.

5 Conclusion

In this thesis, previous fault localization studies that utilized causal inference techniques were analyzed and examined for flaws. The primary flaw that was focused on was the fact that the previous studies were inspired by Gated Single Assignment (GSA) form, but lacked the ability to translate Java programs into true GSA form. Although the previous studies argued that their tools' implementations resulted in the same outcome of pure GSA form, doubts were formed that questioned whether or not this were entirely true. Thus, in this thesis a new tool was introduced that can translate numerical Java programs into equivalent numerical Java programs that are in pure GSA form. By nearly replicating the empirical experiments done in previous studies, the new tool was compared to the previous iterations, and it was found that the new tool greatly outperformed them when locating the faulty variables. To reiterate the results, the tool from this thesis was able to correctly rank the faulty variable as the most suspicious variable 57.5% of the time, whereas the previous tool was only able to correctly rank the faulty variable as the most suspicious variable 14.7% of the time. Also, even when the tool from this thesis did not suspect the faulty variable as the most suspicious variable, it still labeled it within the top 7% of suspicious variables 90.8% of the time; whereas the previous tool only labeled the faulty variable within the top 7% of suspicious variables 35.3% of the time. This shows that the doubts raised by the "inspired" versions of GSA were valid, as the pure form of GSA clearly resulted in superior fault localization results.

6 Future Work

6.1 Array SSA Form

As mentioned earlier, along with implementing an "inspired" version of GSA form, Sheng's tool also implemented an "inspired" version of Array SSA form [20]. However, for the tool from this thesis, a real version of Array SSA form should be implemented, just like how a real version of GSA form was implemented. This would allow this tool to include arrays in the fault localization process. Meaning, the tool would be able to detect if an array assignment statement were the cause of a fault, and it would be able to add array variables to causal relationships for other variables; which would allow array variables to be adjusted for, to potentially eliminate confounding bias.

Along with extending the tool to account for array variables, other non-primitive types should be added to the implementation as well (such as strings and class objects). String variables can follow generic GSA form, so those just need to be considered during the fault localization process. As for class object variables, Array SSA form can actually be extended to work for those types of variables [17]. Thus, by figuring out how to implement Array SSA form in a future iteration of this tool, it would make the addition of object variables far less difficult, since object variables can be accounted for using an extension of Array SSA form.

6.2 Java Project Functionality

The tool introduced in this thesis only worked on individual Java programs. However, when dealing with real Java software systems, Java files will often be used in cohesion with one another within a Java project. Thus, a future iteration of this tool should be developed that can translate an entire Java project into GSA form, and gather a causal map for all variables within the entire project. This would allow the tool to be testable on a larger scale, which could permit an empirical study on a real Java software system, or something comparable. For example, in [14], Küçük's tool was tested on large Java projects included in the Defects4J library [5]. A future iteration of this tool can be used in a similar empirical study, but this time with a real GSA form implementation.

Appendix A

GSA Source-to-Source Compiler Tutorial

This appendix will explain how to install and run the tool implemented in this thesis. The detailed steps for installation are displayed below:

(1) Clone the GitHub repository: `https://github.com/otraben/GSACompiler` [21].

(2) Open the repository in an Eclipse project [10]. Download Eclipse IDE if not already installed.

(3) Locate the "Main.java" file, which is in the directory: *src/gsa*.

(4) Before running the "Main.java" file:

- Click *Run -> Run Configurations...*
- Click on *Main*
- Click on the *Arguments* header
- Under *Program arguments*, click on *Variables...* and add: "${file_prompt}"

(5) Run the "Main.java" file under the run configuration that was just made. A window will open up to select an input program. Traverse to the directory: *src/tests*. Select any desired file to be translated into GSA form.

- If you would like to add your own test program to the project, make sure to insert it into the *src/tests* directory.

(6) An output directory will be created for the test program: *src/outputs/[name of test program]_Output*.

(7) The output directory will contain the following files:

(a) ***[name of test program]*.java**: Original test Java program translated into GSA form.

(b) ***[name of test program]*_Faulty.java**: Original test Java program translated into GSA form, but the class name, constructor names, and class object declarations all have "_Faulty" appended to the end of them. This file is used to have faults manually inserted into it.

(c) ***[name of test program]*Map.txt**: Causal map for all of the variables in the test program.

(d) **FaultLocalizationTester.java**: File used to run the non-faulty and faulty Java programs, and produce all of the necessary files for fault localization calculations.

(e) ***[name of test program]*.R**: An R file that will create a table of all GSA variables and their corresponding confounding variables (i.e. all of the variables from the causal map).

(f) **RFCIcode.R**: An R file that contains the implementation of *CounterFault.*

(8) Next, the non-faulty and faulty Java files need to be modified to include "record-output" statements at the location of where the developer desires outputs to be recorded. For example, let's say the developer wants the output of the program to be a variable X that is returned at the end of a function $F()$. For the non-faulty file, the following line would need to be manually inserted above the return statement: *Output.recordProgramOutput("[name of test program]",* X*, false);*. Whereas for the faulty file, the following line

would need to be manually inserted above the return statement: *Output.recordProgramOutput("[name of test program]", X, true);*. The true and false values simply inform the function whether it is being called on from the faulty or non-faulty program, respectively.

(9) Next, a fault needs to be inserted into the faulty program. Where you decide to insert a fault is up to you, but in order for it to work with *CounterFault*, it must be on a numerical assignment statement. You can utilize the "Fluky.java" functions that exist within the repository to create faults, or you can create faults in your own way.

(10) Moving on, the "FaultLocalizationTester.java" file needs to be—potentially—modified before running all of the test executions. By default, the file runs the *main* function for the faulty and non-faulty files, but if you are interested in a different method, modify the code to call on that method instead for both the faulty and non-faulty programs.

(11) Now, the "FaultLocalizationTester.java" can finally be run.

(12) Once all of the test executions complete, the output directory will be updated to contain the following additional files:

  (a) **output.txt**: This is the file that is written to during all run-time executions. All variable-value assignments are printed into this file, as well as all output recordings.

  (b) **newoutput.txt**: This file is created using the "output.txt" file. It essentially takes all of the variable-value assignments and formats them into a table that maps each variable to their corresponding values for each test execution.

   (c) **outY.txt**: This file is created using the "output.txt" file. It compares the output from the non-faulty version and compares it with every output from the faulty version. Using these comparisons, either a 1 (which implies a fail) or a 0 (which implies a pass) is appended to the file. Thus, the file contains an outcome result for each test execution.

(13) Next, open up an R environment and run the *[name of test program]*.R file to create the causal adjustment table.

(14) Then, run the RFCIcode.R in the same R environment. It will utilize the "newoutput.txt" file and the "outY.txt" file within it's calculations.

(15) This will result in a CSV file called: **result_secMin_2_p0.75_100tests.csv**. This file will include the suspiciousness scores of all GSA variables from the test program in decreasing order.

References

[1] SciMark 2.0. Available at https://math.nist.gov/scimark2/.

[2] Apache Commons Math 3.6.1. Available at https://commons.apache.org/proper/commonsmath/download_math.cgi.

[3] ANTLR. Available at https://www.antlr.org/.

[4] George K. Baah, Andy Podgurski, and Mary Jean Harrold. Causal inference for statistical fault localization. *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 73–84, July 2010.

[5] Defects4J. Available at https://github.com/rjust/defects4j.

[6] Jiacheng Ding. Causal inference based fault localization for python numerical programs, 2018.

[7] A. Gopnik and C. Glymour. Causal maps and bayes nets: A cognitive and computational account of theory-formation. *In P.Carruthers, S. Stich, M. Siegal (Eds.) The cognitive basis of science*, Cambridge(Cambridge University Press): 117–132, 2002.

[8] Java Grammar. Available at https://github.com/antlr/codebuff/blob/master/grammars/org/antlr/codebuff/Java.g4.

[9] Miguel A. Hernán and James M. Robins. *Causal Inference: What If.* Boca Raton: Chapman Hall/CRC, December 2020.

[10] Eclipse IDE. Available at https://eclipseide.org/.

[11] JAMA. Available at https://math.nist.gov/javanumerics/jama/.

[12] Aarnav Jindal. Transpilers : Source-to-source compilers. July 2019.

[13] Yigit Kucuk, Tim A. D. Henderson, and Andy Podgurski. Improving fault localization by integrating value and predicate based causal inference techniques. 2021. doi: 10.48550/ARXIV.2102.06292. URL https://arxiv.org/abs/2102.06292.

[14] Yiğit Küçük. *Causal Basis of Value-Based Statistical Fault Localization.* PhD thesis, Cleveland, Ohio, 2022.

[15] W. Masri and R. A. Assi. Prevalence of coincidental correctness and mitigation of its impact on fault localization. *ACM transactions on software engineering and methodology (TOSEM)*, 23(1):8, 2014.

[16] Brady Neal. *Introduction to Causal Inference from a Machine Learning Perspective*. December 2020.

[17] Lots of authors. *Static Single Assignment Book*. May 2018.

[18] Andy Podgurski and Yiğit Küçük. Counterfault: Value-based fault localization by modeling and predicting counterfactual outcomes. pages 382–393, 2020. doi: 10.1109/ICSME46990.2020.00044.

[19] Nicklaus Roach. Gated single assignment instrumentation and fault localization for numerical java programs. Master's thesis, Cleveland, Ohio, 2019.

[20] Jian Sheng. Value-based fault localization in java numerical software with causal inference technique. Master's thesis, Cleveland, Ohio, 2019.

[21] GSA Source to-Source Compiler for Java Programs. Available at `https://github.com/otraben/GSACompiler`.

[22] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42 (8):707–740, 2016. doi: 10.1109/TSE.2016.2521368.