A case study in Non-Functional Regression Verification

Mohit Bangalore Venkatesh
Abstract

A case study in Non-Functional Regression Verification

Mohit Bangalore Venkatesh

In the process of software evolution, it is an important concern to prevent the introduction of unwanted behavior or bugs (known as regressions) due to updates or when a new feature is implemented in the software. Regression testing is one common solution to identify regressions; a complementary approach that has the same goals as regression testing, but applies methods from formal verification is called regression verification. Usually, both regression testing and verification only consider functional properties, e.g., the results produced by a program. In this thesis, non-functional regressions are considered, in particular detection of modifications that adversely affect program run-time.

Applying the method of regression verification, it is shown that two versions of a program produce the same output for all inputs, as well as verified that the two versions are equal with respect to run-time. For the latter step, programs are instrumented to count the number of executed instructions (weighted by the cost of instructions); those counters can then be compared using functional regression verification. Experiments show that this combination of functional regression verification and instrumentation can effectively analyze non-functional equivalence properties for a range of example programs. In the experiments, the REVE tool, is used to determine functional equivalence after instrumentation.
Contents

1 Introduction  2  
  1.1 Motivation ................................................. 2  
  1.2 Regression ............................................... 3  
  1.3 Equivalence .............................................. 3  
  1.4 Introductory Examples ................................. 6  
  1.5 Thesis Goals ............................................. 9  

2 Background  12  
  2.1 Grammar of the considered language .................. 12  
  2.2 Partial(Functional) Equivalence of Programs ........... 18  
  2.3 Technique for checking program equivalence using Horn clause  19  
  2.4 Related work ........................................... 21  

3 Checking Non-Functional Equivalence  24  
  3.1 Abstract Execution time ................................ 24  
  3.2 Comparing Execution times .............................. 26  
  3.3 Translating Nonfunctional Equivalence to Functional Equivalence  28  
  3.4 Instrumentation ........................................ 29  
  3.5 Program Slicing ......................................... 30  

4 Experiments  34  
  4.1 Setup of the Experiment ................................ 34  
  4.2 Arrangement of the files ................................. 35  
  4.3 Asserting the results ................................... 36  
  4.4 Selection of Benchmarks ................................ 37  
  4.5 Differences between the Programs ........................ 38  
  4.6 Results ................................................. 39  

5 Discussion and Conclusions  41  
  5.1 Future work ............................................ 42  

6 Appendix  46
Chapter 1

Introduction

This chapter gives an insight into the motivation for the thesis work, a little explanation about equivalence as to how we decide that two programs are equal or unequal with some examples. Finally the thesis goals.

1.1 Motivation

In the process of software evolution, we want to prevent the unwanted behavior known as regressions that arise due to some updates or when a new feature is implemented in the software. The primary concern is to avoid this behavior, especially during optimizations or when fixing defects. They may cause severe consequences in the late stages of software development [10].

Regression testing is the solution to suppress the regressions that appear during the software evolution. A complementary approach that performs the same goals as regression testing with methods from formal verification is called regression verification. A formal proof of equivalence is established between the programs. We use REVE (available at http://formal.iti.kit.edu/improve/) tool in finding out the equivalence, and the run-time of the tool (abstract execution time) after the programs are instrumented[15]. We are attempting to show that the two versions of the program produce the same output for all inputs, and the next step will be to verify that these two versions are equal with the runtime-wise [10].

However, regression verification may not completely replace regression testing as regression testing has well-known benefits. If the regression verification is made automatic which can be an extra means for the quality assurance, then additional expenses such as to develop and maintain the test suite can be prevented [10]. Several techniques are available to check the program equivalence. See section 2.4.
1.2 Regression

Unwanted behavior that occurs due to the updates or a new feature is implemented in the software is called regressions. Consider a program to add two numbers using functions. See Figure 1.1.

Program_1 has two functions which add two variables and return the result to the calling function. However, in the second function addWithoutTemp, I have used the variable a to add variable b and save the result in the variable a. When I print the variables a and b, I realize the value of variable a has changed because of the function addWithoutTemp.

Program_2 has another function addNewReg which will use the altered value of the variable a to add the numbers, and the result would vary, since the value of a has changed because of the function addWithoutTemp. We can consider the update or new feature in the function addWithoutTemp and the unwanted behavior as the result that is produced by the function addNewReg.

1.3 Equivalence

Programs were written in ANSI-C which is a subset of language C. These have a function name with arguments as integers, and the function is expected to return a variable at the end of the function as in the C language. Two programs are considered to be partially equivalent when they produce the same output for any given inputs. See Figure 3.3.

The two programs are written in a different way such that when any inputs are given to the programs they produce same outputs which mean they are partially equivalent [10]. Each program has a function name which is of type int, and every function has arguments of type int with a return statement at the end of the program.

The next version of the program contains reversing the contents of an array, and the two programs are written using a different way, but when any inputs are given to the programs they produce the same outputs which means they behave equally. See Figure 1.3. They have a function name which is of type void, and every function has arguments of type int with a return statement at the end of the program.
```c
int main()
{
    int a = 5, b = 10, c = 0;
    c = addWithTemp(a, b);
    a = addWithoutTemp(a, b);
    printf("%d", a);
    printf("%d", b);
    return 0;
}

int addWithTemp(a, b)
{
    int t = 0;
    t = a + b;
    return t;
}

int addWithoutTemp(a, b)
{
    a = a + b;
    return a;
}

int addNewReg(a, b)
{
    a = a + b;
    return a;
}
```

(a) Program to add numbers using function program_1

(b) Program to add numbers using function program_2

Figure 1.1: Programs to add two numbers using function
int nested_while_1
(int x, int g){
    int i = 0;
    while (i < x){
        i = i + 1;
        g = g - 2;
        while (x < i){
            x = x + 2;
            x = x - 1;
            g = g + 1;
        }
    }
    return g;
}

(a) Basic version of Nested while program_1

int nested_while_2
(int x, int g){
    int i = 0;
    while (i < x){
        i = i + 1;
        g = g - 1;
        while (x < i){
            x = x + 1;
            g = g + 1;
        }
    }
    return g;
}

(b) Basic version of Nested while program_2

Figure 1.2: The basic version of Nested while programs

void reverse_array_1
(int n, int *a){
    int i = 0;
    int j = n - 1;
    int t = 0;
    while (i < j){
        t = a[i];
        a[i] = a[j];
        a[j] = t;
        i++;
        j--;
    }
}

(a) Basic version of reverse array Program_1

void reverse_array_2
(int n, int *b){
    int i = 0;
    int j = n - 1;
    int t = 0;
    while (i < (n/2)){
        t = b[i];
        b[i] = b[j];
        b[j] = t;
        j--;
        i++;
    }
}

(b) Basic version of reverse array Program_2

Figure 1.3: The basic version of Reverse Array programs
```c
int Nested_while_1(int x, int g){
while (i < x){
    C = C + cost_>
    i = i + 1;
    C = C + cost_++;
    g = g - 2;
    C = C + 2 * cost_--;
    g = g + 1;
    C = C + cost_++;
while(x < i){
    C = C + cost_>;
    x = x + 2;
    C = C + 2 * cost_++;
    x = x - 1;
    C = C + cost_--;
    g = g + 1;
    C = C + cost_++;
}
    C = C + cost_>
}
    C = C + cost_>
return C;
```
int Nested_while_2(int x, int g) {
  while (i < x) {
    C = C + cost_>; 
    i = i + 1;
    C = C + cost_++;
    g = g - 1;
    C = C + cost_--;
    while (x < i) {
      C = C + cost_>; 
      x = x + 1;
      C = C + cost_++;
      g = g + 1;
      C = C + cost_++;
    }
    C = C + cost_>; 
  }
  return C;
}

Figure 1.5: Instrumented version of Nested while2 Program_2

int reverse_array_1(int n, int *a) {
  while (i < j) {
    C = C + cost_<;
    t = a[i];
    a[i] = a[j];
    a[j] = t;
    i++;
    C = C + cost_++;
    j--;
    C = C + cost_--;
  }
  C = C + cost_<;
  return C;
}

Figure 1.6: (a) Instrumented version of Reverse array1 Program_1
int reverse_array_2(int n, int *a) {
    while (i < (n/2)) {
        C = C + cost_/ + cost_<<;
        t = a[i];
        a[i] = a[j];
        C = C + cost_==;
        a[j] = t;
        j--;
        C = C + cost_--;
    }
    C = C + cost_/ + cost_<<;
    return C;
}

Figure 1.7: (b) Instrumented version of Reverse array2 Program_2

void bubble_sort_1(int *a, int n) {

    int j = 0;
    int i = 0;
    int t = 0;

    while (i < n - 1) {
        while (j < n - i - 1) {
            if (a[j] > a[j + 1]) {
                t = a[j];
                a[j] = a[j + 1];
                a[j + 1] = t;
            }
            j++;
        }
        i++;
    }
}

Figure 1.8: (a) Basic version of Bubble sort1 Program_1

produce same outputs. See Figure 1.3. When these programs are instrumented, then we increment for every operation such as condition and looping. See Figures 1.6 and 1.7.

These programs are instrumented depending on the operations involved, which are explained in the later section. See section 3.4. The variable C is first declared and initialized to zero. Then it is incremented for the comparison operation in the while loop, and incremented again depending on the operations involved. Finally, increment variable C, while program exits the while loop to maintain the consistency.

The next program is bubble sort, and both the basic versions will produce the same outputs for any given inputs. The difference in the two programs is that in the program_1 the inner while loop iterates until it reaches the value of the difference between the last item and the count value from the outer while loop; whereas in
```c
void bubble_sort2(int *a, int n){
    int j = 0;
    int i = n - 2;
    int t = 0;
    while (i >= 0){
        while (j <= i){
            if (a[j] > a[j + 1]){
                t = a[j];
                a[j] = a[j + 1];
                a[j + 1] = t;
            }
            j ++;
        }
        i --;
    }
}
```

Figure 1.9: (b) Basic version of Bubble sort2 Program_2

the program_2, the inner while loop iterates until it is equal to lesser than the count value from the outer while loop. See Figures 1.8 and 1.9.

The bubble sort programs are instrumented depending on the arithmetic, conditional and looping operations. Finally, we return the C variable to assert the two programs. See Figures 1.10 and 1.11. In the above examples, the instrumentation of programs is done depending on the operations and looping that are involved.

### 1.5 Thesis Goals

Define a primary form of code instrumentation that makes it possible to detect non-functional regressions (in particular, Abstract Execution Time) using methods for functional regression verification. We have chosen a variety of programs from the C language. The list of programs varies from simple calculations such as a calculator, Fibonacci series, arrays to sorting algorithms, search an element in an array. We have a program that also has recursion. For each program, we have two versions of the program that are written in a different way but still produces the same output, so instrumentation is used to evaluate the abstract execution costs of the different variants.

Investigate the feasibility of applying existing automated regression verification techniques to abstract timing costs(Abstract execution time), using a case study. The feasibility means to know how long the REVE tool takes to find the equivalence between the programs after the programs are instrumented.

The theme of the thesis is to explore how the methodology can be extended to non-functional property of a program, in particular for the analysis of abstract exe-
int bubble_sort_1(int *a, int n){
    while(i < n - 1){
        C = C + cost,-- + cost,<;
        while(j < n - i - 1) {
            C = C + 2 * cost,-- + cost,<;
            if(a[j] >= a[j + 1]){
                C = C + cost_R + cost,++ + cost,<;
                t = a[j];
                a[j] = a[j + 1];
                C = C + cost,++ + cost,==;
                a[j + 1] = t;
                C = C + cost,++ + cost,==;
            }
            C = C + cost,++ + cost,<;
            j++;  
            C = C + cost,++;
        }
        C = C + 2 * cost,-- + cost,<;
        i++;  
        C = C + cost,++;
    }
    return C;
}

Figure 1.10: (a) Instrumented version of bubble1 Program_1

int bubble_sort_2(int *a, int n){
    while(i >= 0) {
        C = C + cost,<;
        while(j <= i) {
            C = C + cost,<;
            if(a[j] >= a[j + 1]) {
                C = C + cost_R + cost,++ + cost,<;
                t = a[j];
                a[j] = a[j + 1];
                C = C + cost,++ + cost,==;
                a[j + 1] = t;
                C = C + cost,++ + cost,==;
            }
            C = C + cost,++ + cost,<;
            j++;  
            C = C + cost,++;
        }
        C = C + cost,<;
        i++;  
        C = C + cost,++;
    }
    C = C + cost,<;
    return C;
}

Figure 1.11: (b) Instrumented version of bubble sort2 Program_2
cution time, which is essential when developing embedded software. For instance, we analyze whether modifications of a program have any adverse effect on the execution time (abstract) of the program, or verify that the abstract execution time does not get worse as a result of modifications. To this end, a case study will be conducted on how (simple) imperative programs can be rewritten (instrumented) so that non-functional properties are mapped to functional properties.
Chapter 2

Background

This chapter contains the background such as the programming language that is considered for the instrumentation with a detailed description of the programming language and the operators involved. Finally, the program equivalence, in particular, the partial equivalence, and the available techniques for checking program equivalence.

In the basic version of the program, it has been proved that the same output is produced for any given inputs [10]. An automated tool to check Regression Verification was developed at Uppsala University (with the Karlsruhe Institute of Technology). Now we use REVE tool to instrument the programs and find the runtime of the instrumented programs and to address the nonfunctional properties of the program, in particular, the abstract execution time.

While working on Embedded systems especially on the real-time systems, the most important thing is the abstract execution time. We use the regression verification as a tool to observe the execution time(runtime), in particular, the relative execution time rather than the absolute execution time between the programs.

2.1 Grammar of the considered language

The language used to write the programs is a subset of C language. This language can only handle data type of Integers and pointer to integers. Furthermore, it can only handle the \texttt{while} loop. If programs are written using a \texttt{for} loop, then we will have to rewrite them using a \texttt{while} loop. The tool can also handle an array of integers. Every program must have a specification with a function. Each function has a type, id with statements and a body.

The type can only be an integer which is expressed as int in C language or pointer to an integer *int. For example, if a variable is declared as an int x, then id would be
```c
int Calculate(int *a, int n) {
    int b = 5;
    int *c = &b; // pointer c is defined
    b = *a + n;
    *c++; // pointer c is incremented
    b--; // variable b is decremented
    return n; // variable n is returned using return statement
}
```

Figure 2.1: Example of a program

x, and type would be an integer which is represented as an int. The body contains declaration with the expression, condition, and a statement. Every statement should be declared using a semicolon at the end. See Figure 2.1.

The table gives a broader view of the language, see Table 2.1. Each program defined has a specification and a function. The specification can be a literal thing which is evaluated by the REVE tool as a comment. Everything that is in between "" is considered to be literal.

The function consists of the type that can be void or a type id, and the body of the program. The type can be a data type as an integer or a pointer to an integer. If it is a pointer to an integer, we must use an asterisk. See Figure 2.1.

Referring to the figure 2.1, the function has a name called Calculate with arguments as an array *a and an integer variable n, see line number 1. Statements that are declared in the function Calculate ends with the semicolon, see line numbers 6, 7, and 8. The pointer c is declared and used in the program, see line numbers 4 and 7.

The body contains variable declaration with the statements that specify the operation involved with a return statement ending with a semicolon. All the statements and variables that are declared must end with the semicolon. See Figure 2.1 and line number 10.

The variable declaration consists of type Id that can be an integer which is represented as int and quotes with "" are literal, that equals to an expression. The statement which is declared can be incremented with ++ ending with a semicolon, and decremented with -- ending with a semicolon. The statement can also be a pointer with the asterisk symbol = equals an expression. See Figure 2.1. The variables b and c are incremented and decremented using ++ and -- respectively. See line numbers 6, 7, and 8.

The statement can also be arrays which are represented by an expression with "[" and the integer can be inside ending with "]" that equals = to an expression of an integer ending with a semicolon. The expression can also be an array of integers that can be declared using "{" and can be filed with the integer list separated by
Table 2.1: List of operators in C language

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>Semicolon</td>
</tr>
<tr>
<td>=</td>
<td>Equals</td>
</tr>
<tr>
<td>,</td>
<td>comma</td>
</tr>
<tr>
<td>*</td>
<td>Asterisk</td>
</tr>
<tr>
<td>]</td>
<td>Right square bracket</td>
</tr>
<tr>
<td>[</td>
<td>Left square bracket</td>
</tr>
<tr>
<td>)</td>
<td>Right parenthesis</td>
</tr>
<tr>
<td>(</td>
<td>Left parenthesis</td>
</tr>
<tr>
<td>}</td>
<td>Right curly brackets</td>
</tr>
<tr>
<td>{</td>
<td>Left curly brackets</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>Quotation mark</td>
</tr>
<tr>
<td>++</td>
<td>Increment</td>
</tr>
<tr>
<td>--</td>
<td>Decrement</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>\</td>
<td>Division</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>a[ ]</td>
<td>An array of integers</td>
</tr>
<tr>
<td>*a</td>
<td>A pointer a</td>
</tr>
<tr>
<td>&amp;</td>
<td>AND operation</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>;</td>
<td>Termination symbol</td>
</tr>
<tr>
<td>,</td>
<td>Concatenate symbol</td>
</tr>
<tr>
<td>!</td>
<td>Negation</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater</td>
</tr>
<tr>
<td>&lt;</td>
<td>Lesser</td>
</tr>
<tr>
<td>=</td>
<td>Equals or definition symbol</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Lesser than equal to</td>
</tr>
<tr>
<td>==</td>
<td>Equal to</td>
</tr>
<tr>
<td>if</td>
<td>IF Conditional statement</td>
</tr>
<tr>
<td>else</td>
<td>ELSE Conditional statement</td>
</tr>
<tr>
<td>While</td>
<td>WHILE loop statement</td>
</tr>
</tbody>
</table>
### Table 2.2: Grammar of ANSI-C language

Program ::= { Specification | Function }

Specification ::= /*@*/ ("rel_out") Condition "/@*/

Function ::= (Type | "void") Id "(" [ Type Id { "," Type Id } ] ")" "{" Body "}"

Type ::= "int" | "int" "*"

Body ::= { VarDecl } { Statement } { "return" Expression ";" }

VarDecl ::= Type Id [ "=" Expression ];

Statement ::= Id "=" Expression ";" |
             Id "++" ";" |
             Id "–" ";" |
             *Expression "=" Expression ";" |
             Expression ["[" Expression "]" "=" Expression ";" ] |
             Id "=" Id "(" [ Expression { "," Expression } ] ")" ";" |
             "if" "(" Condition ")" BlockOrStatement |
             [ else BlockOrStatement ] |
             "while" "(" Condition ")" BlockOrStatement

BlockOrStatement ::= "{" { Statement } "}" | Statement

Expression ::= [ "\other" ] Id |
              [ "\other" ] "\result" |
              NumberLiteral |
              "-" Expression |
              Expression ( "+" | "-" | "+" | "*" | "/" | "%") Expression |
              "(" Expression ")"

Condition ::= "!" Condition |
              Condition ( "&" | "|" | "||" ) Condition |
              Expression ( "==" | "!=" | "<" | "<=" | ">" | ">=" | ">=" ) Expression |
              "(" Condition ")"

Id ::= [A-Za-z0-9]+ |
    NumberLiteral ::= [0-9]+
```c
int Calculate(int a, int n, int)
{
    int d[5] = {1,2,3,4,5}; //array d has five elements
    int c = 0;
    int l = 5;
    int m = 7;
    int k = 0;
    int t = 0;
    if (l > m) //condition used in the if statement
    {
        k = k + 1;
        t = t + 1;
    }
    else
    {
        k = k - 1;
        t = t - 1;
    }
    return n;
}
```

Figure 2.2: Example of a program with a function and if-else condition

The language has conditional statements like if-else which are keywords as in language C. If with a condition and else should have blocks or the statements. The else part is optional. See Figure 2.2. This function Calculate has an if-else condition with statements and the body. See line number 11 for the if condition and the line numbers 12 to 15 for the body of the if condition. See line number 16 for the else condition and the line numbers 17 to 20 for the body else condition.

While is a keyword in C and while loop has a condition with a block or statements ending with a semicolon. See Figure 2.3. The body of the while loop is from line numbers 12 to 16. There are no immediate exit statements such as the break or the continue as in C language [10].

The expressions have operations such as addition, subtraction, division, multiplication, and modulus represented as +,−,/,∗, and % respectively. See Figure 2.3. See line numbers 13, 14, 15, and 20 for the operation used between variables k, t, and c.

The conditions that are used for the if and while loop operation has Boolean conditions such as not equal to, equal to, less than, less than equal to, greater than, and greater than equal to, which is represented as !=, ==, <,<=, >, and >= respectively. All the conditions are used between "(" and ")". See Figure 2.3. See line numbers 11 and 18 for the Boolean conditions that are used in the while loop.
```c
int Calculate(int a, int n, int)
{
    int d[5] = {1, 2, 3, 4, 5};
    int c = 0;
    int l = 5;
    int m = 7;
    int k = 0;
    int t = 0;

    while (l >= m) // condition used in the while loop
    {
        k = k + 1;
        t = t / 1;
        c = c * 2;
    }

    while (l == m) // condition used in the while loop
    {
        k = k - 1;
        t = t - 1;
    }

    return n;
}
```

Figure 2.3: Example of a program with a while loop

The identifiers usually will start with the digit or a letter as in the language C. The specifications rel_out are not used as statements in the program but are used as a comment to assert the two programs. The expression has \other refers to the other program which is program_1 and equals with \other refers to the other program which is program_2.

Here specification is defined using "*/ @@*/" in which it contains the rel_out condition to assert both the programs.

/ * @ rel_out condition @ */

The rel_out condition is applied to the output, Which means we try to assert output produced by program_1 with the output produced by program_2, as in C language assert(result1,result2).

The result1 refers to the output produced by program_1, and result2 refers to the output produced by program_2. For simplicity purpose and as per the grammar of the language C, we must have the return statement at the end of the program only [10].

To assert the programs, a command is used in REVE tool which is given below.

/ * @ \result == \other \result@ */

See Figure 2.4. See line numbers 1, 2 in program_2. The \result which is on the L.H.S of the statement refers to the output produced by the program_1 and \result which is on the R.H.S of the statement refers to the output produced by the
2.2 Partial(Functional) Equivalence of Programs

The programs that are considered for equivalence are written in ANSI-C language. All programs are considered to be deterministic, which means the program runs starting in the same state will also terminate in the same state. We have assumed all other variables which are used in the programs are initialized. The starting point for the program is the function and always ends with the \texttt{return} statement [10].

We have considered two programs $P_1$ and $P_2$. The program inputs are referred as $i_1$ and $i_2$ and results that are returned as $r_1$ and $r_2$ for the programs $P_1$ and $P_2$ respectively.

Formally we can define the formula as

\[ P_1 \simeq P_2 \iff \forall i_1, i_2. (i_1 = i_2 \rightarrow \text{wlp}(P_1; P_2, r_1 = r_2)) \]  \hspace{1cm} (2.1)

Two programs $P_1$ and $P_2$ are said to be partially equivalent, if and only if the programs, whenever they terminate for the same inputs $i_1 = i_2$, they produce the same outputs $r_1 = r_2$.

The wlp stands for the weakest liberal precondition. The predicate $\text{wlp}(P, \varphi)$ denotes the $\varphi$ needs to hold before the execution of the program $P$ if execution terminates and the post condition $\varphi$ holds in the final state.

The term liberal means that the program can terminate, or it may not terminate if
it has loops. The \( \text{wlp}(P_1; P_2; \phi) \) can be explained as, if programs \( P_1 \) and \( P_2 \) have started in the same state and terminates, then post condition \( \phi \) holds in its final state. \( \text{wlp}(x:=y+1; x = 10) \) is as good as saying \( y + 1 \) will be equal to 10 when \( y \) holds a value of 9. The \( \phi \) is \( x = 10 \) and \( P_1 \) and \( P_2 \) is \( x:=y+1 \).

2.3 Technique for checking program equivalence using Horn clause

Programs \( P_1 \) and \( P_2 \) are formulated over a disjoint set of program variables such that there is no interference between the program \( P_1 \) and \( P_2 \). The two programs \( P_1 \) and \( P_2 \) are converted into verification conditions by adding coupling predicates. The mutual invariants that describe the reachable states of two loops in the respective program are iterated in a synchronized manner [10].

These generated verifying conditions are represented in the form of Horn constraints and then are solved using ELDARICA or Z3 solvers. There are some algorithms to solve this Horn clauses like predicate abstraction [11, 12] and property-directed reachability [16].

The Horn solver will produce two possible results namely (1) a symbolic solution to the processes Horn clauses. Or (2) a counter example tree that shows no solution for the Horn clauses that are generated does not exist. If a symbolic solution to the Horn clauses is generated, then it means the two programs \( P_1 \) and \( P_2 \) will behave equally, and if the tool is successful in creating the counter example for the Horn clauses, then it means the two programs \( P_1 \) and \( P_2 \) will not behave equally [10]. See figure 2.5.

The variables \( z_1, i_1 \) and \( z_2, i_2 \) belongs to program_1 and program_2 respectively. \( S_{g_1} \) and \( S_{g_2} \) represent the body of program_1 and program_2 respectively. See Figure 2.6. When these programs are run on the REVE tool, they are converted into verification conditions(VC). See Figure 2.7.

These verification conditions are converted into Horn clauses. See Figure 2.8. These Horn clauses are solved in the Horn solver such as Z3 or ELDARICA. If coupling predicates are found then the program_1 and program_2 are partially equivalent. See Figure 2.9.

In the case of coupling predicates do not exist, and the REVE tool is successful in providing a counter example, then program_1 and program_2 are not partially equivalent. See Figure 2.10.

The name of the tool is called REVE. This tool REVE is [http://formal.iti.kit.edu/improve/](http://formal.iti.kit.edu/improve/) used to find the equivalence between the programs [10]. See Figures 3.3 and 1.3.
Figure 2.5: Brief description of the technique used in Horn clause method

Figure 2.6: Basic version of while loop programs
false $\leftarrow z1 \leq 0 \land z2 \leq 0 \land z1 = z2$
false $\leftarrow z1 > 0 \land z2 > 0 \land z1 = z2$
false $\leftarrow z1 > 0 \land z2 \leq 0 \land z1 = z2$
false $\leftarrow z1 \leq 0 \land z2 > 0 \land z1 = z2$

Figure 2.7: Verification conditions of program_1 and program_2

$Sg_1(z_1, i_1) \leftarrow z_1 \leq 0$
$Sg_1(z_1, i_1) \leftarrow z_1 > 0 \land Sg_2(z_1, i_1)$
$Sg_2(z_2, i_2) \leftarrow z_2 \leq 0$
$Sg_2(z_2, i_2) \leftarrow z_2 > 0 \land Sg_2(z_2, i_2)$

Figure 2.8: Horn clauses of program_1 and program_2

### 2.4 Related work

There are many different techniques for checking the program equivalence. A tool called RVT [9] is developed by D. Kroening, and it is a bounded model checking tool for the programs that are written using ANSI-C, which is a subset of C. CBMC model checker is used to find the equivalence of two programs. The programs that are written and used in the REVE tool also follows ANSI-C. In any given program for each loop $i$, the user needs to specify a bound $k_i$ on the number of iterations. This process makes CBMC tool to characterize the full set of possible executions restricted by the bounds, given by the user, with the propositional formula $f$.

The tool allows the user to check, the bounds provided by the user are high enough. This process is done by generating special unwinding assertions for each loop [9]. An unwinding assertion for a loop $i$, given $k_i$, is satisfied only if this condition of the loop cannot be true after iterating $k_i$ times. This approach has a limit when the program that consists of functions having a different number of arguments.

Functions that are declared void should not have a return statement.

To prove the equivalence of the static programs Verdoolaege et al. [24, 25] have developed an automated approach. It primarily focuses on the programs containing for loops with array manipulation. And it can also deal with programs that include loop skewing, loop iteration, loop tiling. The approach is completed using a tool called ISA tool that uses programs that are written using ANSI-C. The abstraction of each of the two program is considered to build the dependence graph, and data-flow analysis is applied.

Barthe et al. [8] present a calculus on transformation for reasoning between pro-
grams. The approach is to form a single product program by merging to programs. The rules, procedures and merging properties are guided by the user with the help of existing verification technology known as WHY tool.

Sinz and Post [21] prove the equivalence of two AES chiper implementation by the bounded model checking. This approach is more feasible that involves loops and recursive calls but only limited if the program has small loops of recursive calls. Almedia et al. [2] have verified the correctness of the OpenSSL implementation of the RC4 cipher on reference implementation.

Backes et al. [4] propose to slicing and study analysis to improve the scalability of regression verification. The main idea is to remove the code in both the programs that have no data or control dependencies on the introduced change and show equivalence of the reduced programs by an existing technique like the bounded symbolic execution.

Hawblitzel et al. [13] put forth mutual function summaries and later developed in [14]. SymDiff [19] equivalence checker where the shared summary are supplied by the user and the verification are discharged by Boogie. Usually, loops are encoded as recursion. The BCVerifier tool for providing backward compatibility. Welsch and Poetzsch-Heffter [26] have similar pragmatics.

The SYMDIFF tool was combined with the Houdini invariant generation algorithm to infer coupling predicates for regression verification of memory safety properties. Houdini attempts to construct a consistent set of inductive conjunctive invariants by "brute force" elimination from a pool of candidates built by instantiating a user-specified template. However, time performance data is not reported in [20]. This tool has more resemblance like in our tool where it is automatic and does not expect the user to give variants. This tool SYMDIFF is very close to the tool we have used.
for instrumentation which does not require to give initial predicates from the user.

Programming discipline and a static analysis ensuring that changes in an object-oriented data structure’s implementation are confined and cannot affect its clients other than through specified public methods. Study of equivalence of Java-like programs from the perspective of data encapsulation was presented by Banerjee and Naumann [5, 6]. To address security applications, several program logics (e.g., [3, 7, 22]) have been developed. To prove the logic this requires user-supplied inductive invariants.

A Tremendous amount of work exists on the equivalence checking of hardware logic circuits [17]. It is divided into major groups. One group that builds the product machine of two circuits and traverses the state space such that the corresponding outputs are identical in every reachable state. The other group recognizes that the process induces structural similarity from the incremental nature of the circuit variants and exploit them like functional equivalence, indirect implications, permissible functions and others [23].
Chapter 3

Checking Non-Functional Equivalence

This chapter contains the Abstract Execution time and relative Execution time with the instrumentation. Finally, the program slicing.

3.1 Abstract Execution time

The execution time that has been measured in our thesis can be called abstract time as a metric for comparison effort. It is a simplified model where we have not considered many factors that will affect the execution time such as pipeline, memory, processor and cache. Usually, the execution time depends on some factors such as the number of inputs, program structure or the way the program is written. Every single statement would consume some time or cycle to execute. The table shows the number of increments that are used in the process of measuring the cycle for each instruction, and their respective count. See Table 3.1.

Let us look into an example without a loop. Consider this program that has simple statements which add two variables and returns the result using the return statement. See Figure 3.1. In this program, we can say the execution time depends on the value of the variable \( x \). We may also conclude that the amount of execution time needed by this program or the cycle would be two. That is because it has two arithmetic operations \( ADD \), so the cycle or the cost would be two. See line number 5. See Table 3.1 for the number of cost for any particular operation.

Let us consider another program that has a loop with an array. See Figure 3.2. The number of cycles needed for this program would be \( 2 + 1 + N \times 3 \). The value two is got because the \( while \) loop has a comparison, and the value one is got because it has an increment operator, and the value three is got because it has an arithmetic
### Table 3.1: Operations and their count

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Execution time in cycles or count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reading from an array</td>
<td>2</td>
</tr>
<tr>
<td>Assigning to an array</td>
<td>2</td>
</tr>
<tr>
<td>Recursion</td>
<td>4</td>
</tr>
<tr>
<td>AND operation</td>
<td>3</td>
</tr>
<tr>
<td>Multiplication</td>
<td>3</td>
</tr>
<tr>
<td>Addition</td>
<td>1</td>
</tr>
<tr>
<td>Division</td>
<td>4</td>
</tr>
<tr>
<td>Modulus</td>
<td>2</td>
</tr>
<tr>
<td>Subtraction</td>
<td>1</td>
</tr>
<tr>
<td>Comparison</td>
<td>2</td>
</tr>
<tr>
<td>Increment</td>
<td>1</td>
</tr>
<tr>
<td>Decrement</td>
<td>1</td>
</tr>
</tbody>
</table>

```c
int Addtwointegers(int x){
    int y = 0;
    y = y + x + 1; // two addition, so number of cycles will be 2
    return y;
}
```

### Figure 3.1: Program to add two variables
void CopyValue (int N, int a[i]){
  int i = 0;
  while (i < N) { // comparison cost = 2
    a[i] = N + 1; // addition cost + assigning to an array cost
    i++; // increment cost
  }
}

Figure 3.2: Program to copy a value to an array

operator ADD and an array of assigning cost which sums up as 2 + 1 = 3. See line numbers 5, 6, and 7.

The count value also depends on the value of the variable N and the value of the array. See Table 3.1 for the number of increments. When these statements are converted into an assembly line of code, each line will consume some amount of cycles to execute. From the figure 3.2, the number of cycles required will be 2 + N * 3. The expression seems to be very simple since all other parameters such as cache, pipeline, memory are not considered in the thesis.

3.2 Comparing Execution times

The relative execution time can be defined as the property that program_1 and program_2 are taking an equal amount of cycles to execute or in some cases program_1 and program_2 are taking more or less amount of cycles to run for any input. The partial equivalence of the program can be defined using execution time where we use the number or the count on a variable to say that the two programs would behave equally.

Consider this program which has an array with a while loop. See Figure 3.4. The number of cycles would be 6, if we, consider only the body of the while loop. However, the total number of cycles for the program_1 would be 6 * n + 2. See line numbers 5, 7, 8, 9, and 10.

Consider this program which has an array with a while loop. See Figure 3.5. The number of cycles would be 13, if we, consider only the body of the while loop. However, the total number of cycles for the program_2 would be 13 * n + 2. See line numbers 5, 7, 8, 9, 10, 11, and 12.

The total number of cycles needed for program_2 will be always greater when compared to program_1 for n greater than zero. This is what we have referred to as relative execution time in the thesis.
(a) Basic version of an Array program 1

```
void array_1
    (int N, int n, int *a1)
{
    int i = 0;
    while (i < n){
        a1[i] = N + 1;
        i++;
    }
}
```

(b) Basic version of an Array program 2

```
void array_2
    (int N, int n, int *a2)
{
    int i = n;
    a2[0] = N + 1;
    while (i > 0){
        i--;
        a2[i] = N;
        a2[i] = a2[i] + 1;
    }
}
```

Figure 3.3: The basic version of Array programs

(a) Basic version of an Array program 1 with cost

```
void array_1
    (int N, int n, int *a1)
{
    int i = 0;        C = 0
    while (i < n){    C = 0 + 2 (Comparison)
        a1[i] = N + 1; C = 2 + 1 + 2 (addition + assigning to array)
        i++;
    }  C = 5 + 1 (increment)
    C = 6 + 2 (exit of the loop)
}
```

Figure 3.4: Basic version of an Array program 1 with cost

(b) Basic version of an Array program 2 with cost

```
void array_2
    (int N, int n, int *a2)
{
    int i = n;        C = 0
    a2[0] = N + 1;    C = 1 + 2 (addition + assigning to array)
    while (i > 0){    C = 3 + 2 (comparison)
        i--;            C = 5 + 1 (decrement)
        a2[i] = N;      C = 6 + 2 (assigning to array)
        a2[i] = a2[i] + 1; C = 8 + 1 + 2 + 2
            (addition + reading from array + assigning to array)
    }  C = 13 + 2 (exit of the loop)
}
```

Figure 3.5: Basic version of an Array program 2 with cost
3.3 Translating Nonfunctional Equivalence to Functional Equivalence

To measure the execution, a variable called cycle or $C$ is used. To measure the execution time we count the number of cycles or in other words, we increment the declared variable $C$ for every statement, but the statement in the program should have any operation or condition involved. Depending on the process involved we increment the variable $C$ after the statement. To know the number of increments for any given condition, see Table 4.3.

The table has the variable name used for the instrumentation for every operation or a condition, and the symbol used in the program which exists only in our report for easy understanding. We also have the number of increments for each operation involved in the program. See Figure 3.6.

Here I have taken the same example which we discussed in the previous section. Variable $C$ is used for measuring the number of cycles in the program. First, we have incremented the variable $C$ for the comparison that exists in the while loop, and then for the arithmetic operator and assigning a value to the array. Then increment variable $C$ for the increment operator. And finally, increment variable $C$ while program exits from the while loop. This variable $C$ is returned using a return statement at the end of the program.

We can see this instrumentation on lines 8, 10, 12, and 14 clearly in figure 3.6 for program_1 and 8, 10, 12, 14, and 16 for program_2 respectively. Even though the values selected for the increment is arbitrary, the values are consistent with
the whole process of instrumentation. The only condition that is neglected or not incremented is for the copying of value to the variable or copying a value from one variable to another variable, assuming that it will not consume too much of time.

3.4 Instrumentation

The nonfunctional property refers to the execution time of the program. The REVE tool is used to calculate the abstract execution time of the programs that are partially equivalent. To have a difference between the two programs, we re-write the programs in such a way, but yet they still produce the same result for any given inputs. These programs are built, compiled and run on the GCC compiler, to reassure they produce the same output for any set of given inputs. After the confirmation, we instrument them.

Instrumenting means a measure of counting a variable and increment them depending on the looping and conditions used. We usually call this as cycles. But to make it easy, we use and declare a variable named $C$. This variable is declared in the declaration section of the program and initialized to 0.

After which we try to examine the program and increment the variable $C$, depending on what operations are involved in the program. We increment the variable $C$ after each statement and keep a count of it. One of the requirements of the REVE tool is the return statement should be at the last. Since the program uses the return statement with an integer, then the program should return an integer value.

When checking the program equivalence, both the program will return a common variable at the end of the program to find the equivalence. The tool will add predicates to find whether they behave equally or not. In some examples the REVE tool tries to add a counter-example program to generate the equivalence between the programs, if found then the programs are declared equivalent, and if it cannot provide a counter example, then the programs are declared not equivalent. The maximum time needed for the REVE tool to compute the equivalence is 180 seconds. If the tool could prove the equivalence between the programs, then it gives the time consumed to assert the programs, sometimes REVE tool may also provide an output of timeout, which is because the REVE tool could not provide a counter example for the programs within 180 seconds.

After all, the necessary increment of the variable $C$ is done we use the return statement. This combination of functional properties and the instrumentation is used to answer the non-functional property of the program which is execution time(abstract) in our thesis.
3.5 Program Slicing

Slicing refers to the decomposition of a program into disjoint, that is its independent parts. However, the behavior of the independent parts is unchanged when compared to its original version. The real motivation behind program slicing was to observe, is there any significant change in the runtime of the program after some statements are sliced. Moreover, Horn solver finds it very easy to analyze the program after the programs are sliced.

We will have to choose carefully the set of statements in a program that are not related or connected to the `while` loop operation. Let us consider an example of Fibonacci program, where we comment those lines that will not affect the `while` loop operation. We comment line number 13, 15 and, 16 in program 2. Thereby, we have commented statements that include variables `h, f, g` but not variable `n`. See Figure 3.7.

In our experiments, we have used only static slicing which is most necessary for our experiments. Let us consider another example, and this program has a nested `while` loop. We have commented those statements which will not affect the `while` loop which means by commenting statements that include only variables `g` but not statements that include variables `i` or `x` because it will affect the `while` loop operation. See Figures 3.8 and 3.9.

As previously mentioned if we comment any other statement it may change the `while` loop operation. Consider another example of program slicing. Here we have commented the statements containing variable `x` but not variable `c`. If the statements including variables `c` or `x` are commented then, slicing can have a direct impact on the `if` condition and `while` loop. See Figures 3.10 and 3.11.

The slicing process is mostly used in the maintenance than in the design [1]. There is always a possibility of at least one slice which can be the program itself. This process of isolating few sections of a program can also be applied to algorithms. Slicing is more useful in testing and debugging, where we can eliminate few parts of the program which may not cause the error, so we could focus on main areas of the program that would contain a mistake. The other applications that include slicing are software maintenance, compiler optimization, information flow control and program analysis.

Slicing can also be used in some other applications [18], [1] such as parallelization, program differencing and integration, software maintenance, testing, reverse engineering, and compiler tuning. Program slicing can be applied in program differencing and integration where the primary task is to analyze the old and the new version of a program to determine the new version that represents syntactic and semantic changes in the program. To detect the equivalent behaviors, we compare the program slices of the new and the old version of the programs.
```c
int fib(int n){
    int f = 0;
    int g = 1;
    int h = 0;
    int C = 0;
    int comparisoncost = 2;
    int minusminuscost = 1;
    int pluscost = 1;

    while (n > 0){
        C = C + comparisoncost;
        h = f + g;
        C = C + pluscost;
        f = g;
        g = h;
        n --;
        C = C + minusminuscost;
    }
    C = C + comparisoncost;
    return C;
}
```

(a) Instrumented Fibonacci program showing before slicing

```c
while(n > 0){
    C = C + comparisoncost;
    // h = f + g;
    C = C + pluscost;
    // f = g;
    // g = h;
    n --;
    C = C + minusminuscost;
}
```

(b) Instrumented Fibonacci program (instrumented) showing after slicing

**Figure 3.7: Instrumented Version of Fibonacci Programs before and after slicing**

```c
int Nested_while_1(int x, int g){
    while (i < x){
        C = C + cost_>
    }
    return C;
}
```

**Figure 3.8: Sliced version of Nested while1 Program_1**
```c
int Nested_while_2(int x, int g) {
  while (i < x) {
    C = C + cost_>; 
    i = i + 1; 
    C = C + cost_++;
    // g = g - 1; 
    C = C + cost_--; 
    while (x < i) {
      C = C + cost_>; 
      x = x + 1;
      C = C + cost_++;
      // g = g + 1;
      C = C + cost_++; 
    }
    C = C + cost_>; 
  }
  C = C + cost_>; 
  return C;
}
```

Figure 3.9: Sliced version of Nested while2 Program_2

```c
int if_while_1(int t, int c) {
  if (0 < t) {
    C = C + cost_<; 
    while (0 < c) {
      C = C + cost_<; 
      // x++; 
      C = C + cost_++;
      c = c - 1; 
      C = C + cost_<--; 
    }
    C = C + cost_<; 
  }
  C = C + cost_<; 
  return C;
}
```

Figure 3.10: Sliced version of While-if1 Program_1
During the software maintenance, it is quite difficult to determine the added changes to the program and may lead to the behavioral changes in the other parts, and here we can use static slicing in reducing the program or slice the program and compare each part of the program to the original behavior. Slicing can also be used to parallelize the programs where the slices of the program are executed in parallel, and the output of the slices is spliced together to determine the input-output behavior by still preserving the original program.

This slicing is applied to the considered programs in our thesis, and we have found many results from unknown to timeout where few programs after slicing have shown a behavior of equal and unequal also. See Table 4.6.
Chapter 4

Experiments

This chapter contains the details of the experiments with the setup initialized and arrangement of the files with the selection of benchmarks. Finally, the difference between the programs and the results.

4.1 Setup of the Experiment

The tool does not check the logic of the program. We have to be careful when writing the logic used in the program, to ensure we get desired output[10]. We must consider programs that are written using only the while loop because the tool cannot handle a for loop. If a program is written using a for loop, we shall rewrite them using a while loop. See Figures 1.8 and 1.9. Since it is a usual practice to use a for loop to sort arrays, we have rewritten them using a while loop.

When a program is chosen, we take into consideration of how many arithmetic operations, conditional statements such as if-else and looping statements such as while loop is present. Depending on that we will increment variable C to count the number of cycles.

To measure the cost(instrumentation), we define a variable called C which is also known as the cycle in our thesis. When we increment the variable C during the entry of the while loop, we should also increment the variable C the same number of times during the exit of the while loop to maintain the consistency. However, depending on the condition of the while loop. At the end of the program, we will use the return statement, and the return statement must be the last statement and cannot be used in multiple places [10]. Since integer value is returned, we must have the function type as an integer which is represented as an int.

The two programs that are considered for instrumentation should have the same name with the same number of arguments and with the same type either as an
integer or a pointer to an integer. Variables that are used in the program are assumed 
to be uniquely named.

After all the primary setup has been completed. The instrumented programs are 
run on the tool. The time measured while executing is the CPU clock, which is 
always measured and recorded in seconds. Here the time which is measured is the 
time taken by the REVE tool to find whether the two version of the program behave 
equally or not.

When measuring the time, there is always a difference of plus or minus 3 seconds 
with the initial value. Because the tool is run on a server and tool takes some time 
to start and run depending on whether the tool is busy or free. To be consistent, we 
have chosen an average value while measuring time.

We increment the variable \( C \) depending on the operation and statements in the 
program. However, at the end of the program, we will have to return the variable 
\( C \) using the return statement [10].

The three versions refer to find whether the number of iterations in program_1 is 
greater than \((>=)\) the number of iteration in program_2. Or the number of itera-
tions in the program_1 is equal to \((==)\) the number of iterations in the program_2. 
Or the number of iterations in the program_1 is lesser than \((<=)\) to the number of 
iterations in the program_2.

By this, we achieve a big table that must account for both the instrumented ver-
sion and the sliced version for all the three cases. The time that is measured and 
recorded in the table is not the absolute execution time of the program but the 
runtime tool of the nonfunctional equivalence checker.

### 4.2 Arrangement of the files

To ease our work, I have arranged the file section in the manner discussed below. 
Each folder has the program name which is the main folder. I have three subfolders 
inside this main folder. The first subfolder consists of C file with the two different 
versions of the program as header files and the main C file which is used by the 
GCC compiler to verify both the header programs will produce the same result for 
any given inputs.

The second subfolder consists of the instrumented version of the two programs 
with the results. There are three result files which account to greater than \((>=)\), 
equal to \((==)\) and lesser than \((<=)\) versions for the instrumented programs. The 
third subfolder consists of the sliced version of the two programs with the results. 
There are three result files which account to greater than \((>=)\), equal to \((==)\) and 
lesser than \((<=)\) versions for the sliced programs.
I have written the test cases to run the C program which is the header files, and these programs will be instrumented and sliced in the next part. I declare program_1 and program_2 as two header files and include these file names in the main C file as the header. I declare all the inputs which are the variables for the main C program and use the name of the program_1 and program_2 as a function call, with the inputs that would be sent as arguments. I compile them on GCC compiler to find there are any other errors and then it is run to see whether they produce the same output. The result that is generated is asserted in the main C file to ensure both results that are produced by the function call are the same which is a necessary condition.

I have used this part of writing test cases for all the programs that are instrumented to ensure they will produce the same output for any given inputs. It is a precautionary measure to check both the program will produce the same output for the inputs provided.

4.3 Asserting the results

When we run the programs on the tool we must assert them using rel_out statement to check will they behave equally or not after the instrumentation. We have used the rel_out which is an assert condition in C language and rel_out is used in the second program that is in program_2.

Here three cases arise. One is greater than equal to (\(>=\)), equal to (\(==\)), and lesser than equal to (\(<=\)). All these cases refer to the two programs that are used in the tool. When we want to run the instrumented version of the program we change for the other version in this statement by using appropriate symbols.

This statement is used for the greater than (\(>=\)) version.

\[
/ * @\result >\ other\ \result @ * / \]

This statement is used for the equal to (\(==\)) version.

\[
/ * @\result == \ other\ \result @ * / \]

This statement is used for the lesser than (\(<\)) version.

\[
/ * @\result <\ other\ \result @ * / \]

The value chosen for the instrumentation is always consistent. See Table 4.3. After the instrumented programs are fed to the tool, initially the condition greater than (\(>=\)) is used between the programs, then the time is recorded, and the associated result is copied to the result file which is reserved in the instrumented folder. The results are copied to the result1 text file which is exclusively meant for this operation. Then the condition is changed to equal to (\(==\)) in the rel_out statement and run the programs on the tool. The following time is copied, and the associated result is copied to the result file which is reserved in the folder. The result is copied to the result2 text file meant for this operation. Then the condition is changed to lesser than (\(<\)) in the rel_out statement, and run the programs on the tool. The
Table 4.1: Declaration of Cost variables

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Name used in program</th>
<th>Symbol used</th>
<th>Number of increments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reading from an array</td>
<td>readingcost</td>
<td>cost_R</td>
<td>2</td>
</tr>
<tr>
<td>Assigning to an array</td>
<td>assigncost</td>
<td>cost_==</td>
<td>2</td>
</tr>
<tr>
<td>Recursion</td>
<td>recursioncost</td>
<td>cost_F</td>
<td>4</td>
</tr>
<tr>
<td>AND operation</td>
<td>andcost</td>
<td>cost_&amp;</td>
<td>3</td>
</tr>
<tr>
<td>Multiplication</td>
<td>multiplicationcost</td>
<td>cost_*</td>
<td>3</td>
</tr>
<tr>
<td>Addition</td>
<td>pluscost</td>
<td>cost_++</td>
<td>1</td>
</tr>
<tr>
<td>Division</td>
<td>divisioncost</td>
<td>cost_\</td>
<td>4</td>
</tr>
<tr>
<td>Modulus</td>
<td>moduluscost</td>
<td>cost_M</td>
<td>2</td>
</tr>
<tr>
<td>Subtraction</td>
<td>minuscost</td>
<td>cost_-</td>
<td>1</td>
</tr>
<tr>
<td>Comparison</td>
<td>comparisoncost</td>
<td>cost_&gt;or cost_&lt;</td>
<td>2</td>
</tr>
<tr>
<td>Increment</td>
<td>pluspluscost</td>
<td>cost_++</td>
<td>1</td>
</tr>
<tr>
<td>Decrement</td>
<td>minusminuscost</td>
<td>cost_-</td>
<td>1</td>
</tr>
</tbody>
</table>

following time is recorded, and the associated result is copied to the result file which is reserved in the folder. The result is copied to the result3 text file. The total time allotted for the tool to compute equivalence between the programs is 180 seconds or 3 minutes.

The operating system used is Linux with 64 bit with the dual-core processor, a cache of 1024K bytes, address size of 40 bits physical, and 48 bits virtual. The clock used is the system clock that runs on the processor and time is measured and recorded in seconds.

When we choose to instrument a program, they must return a value [10], in our case it will be the cycle which is called \( C(\text{number of cycles}) \). To distinguish the programs, we use the notation called program_1 and program_2 referring to program1 and program2 respectively. I have named all the examples in this manner discussed above. The names used for the instrumentation is shown in the table. See Table 4.3.

### 4.4 Selection of Benchmarks

A wide range of C programs is used for our case study on Regression Verification. Consider the loop fission program which has arrays. The instrumented version takes twenty-one, eleven and eleven seconds for all the three cases namely greater than, equal to, and lesser than respectively. However, there is no sliced version since commenting any statements will directly affect the while loop. Moreover, the tool produced an output of unknown for all the three cases in the instrumented version. The number of lines in the instrumented version is thirty-seven and fifty-nine in program_1 and program_2 respectively.

The prefetched program that has an array provides the result of timeout for all the
three cases namely greater than, equal to and lesser than. Because the tool could not find out a counter example within 180 seconds. There is no sliced version for this program also. The number of lines in the instrumented version is twenty-two and twenty-four in program_1 and program_2 respectively.

The only program that has recursion in it is named limit. This program took the least amount of time of 0.5 seconds for both the instrumented and sliced versions. The number of lines in the instrumented version is twenty-six and twenty-seven in program_1 and program_2 respectively.

Let us consider some programs that have arrays with sorting. We have insertion sort, bubble sort, and selection sort. All the programs that have sorting do not have sliced version, since commenting any part of the program that includes slicing will have its effect on the while loop operation.

The insertion sort behaves equally for all the three cases greater than(>=), equal to(==), lesser than(<). However, the time taken for greater than(>=), equal to(==), lesser than(<) is twenty-three, twelve, fifty-seven seconds respectively. The highest time taken by the tool to compute equivalence is fifty-seven seconds for the insertion sort. Let us consider a program that has arrays such as linear search and binary search. The only program to give an output as memory out within two seconds is linear search program for the sliced version.

### 4.5 Differences between the Programs

Few programs that are instrumented are shown in appendix section, see section 6. The first set of programs computes some digits in a number. The difference in the programs is the program_1 calculates some digits by dividing it by 10 and repeating it until the number is greater than zero. The program_2 uses loop unfolding where it has several if-else sections to determine the number of digits until the number is not equal to zero. See Figures 6.1 and 6.2. In the next part, these programs are instrumented and sliced.

The second set of programs has a selection sort. The program_1 sorts the contents of the array using a temporary variable but in the program_2, it uses an additional variable called position where the nested while loop is also used. In the outer while loop, the value of i is assigned to variable the position, and in the inner while loop, the comparison is made between the value of position variable and the value of j, rather than the contents of the array as it is done in the program_1, and thereby making this method more efficient. See Figures 6.7 and 6.8. In the next part, these programs are instrumented and sliced.

The third set of programs has to clear the contents of an array. Program_1 assigns value zero to the contents of the array in a while loop until the last element of the array, and the program_2 assigns a value zero to the contents of the array using a
pointer and in a while loop until the last element of the array. See Figures 6.13 and 6.14. In the next part, these programs are instrumented and sliced.

The fourth set of programs has two while loops that are written in a different way. However, returns the same output. Program_1 has two while loops the first loop multiplies the contents of the variable x with 5 and then adds the content with the variable i in the second loop. The program_2 does the same but after the first while loop the variable i is assigned a value one, and then the same procedure is followed as it is followed in program_1. See Figures 6.19 and 6.20. In the next part, these programs are instrumented and sliced.

The fifth set of programs has a binary search. In program_1, the middle element is found by adding the last element and the first element divided by two. And then the target element is found using if else condition and is repeated until the last element is reached. In the program_2, the middle element is obtained by adding the last element and subtraction of the first element from the last element and then dividing it by two. This makes the second program_2 makes more efficient in searching the target element in the array. Then an identical procedure is followed as in program_1 to find the required element in the array. See Figures 6.25 and 6.26. In the next part, these programs are instrumented and sliced.

4.6 Results

The T refers to the time taken by the REVE tool to examine the behavior of the two instrumented programs in seconds. The B refers to the behavior of the programs for the three cases greater than, equal to and lesser than.

T/O represents the time out,— represents that these set of programs cannot be sliced. ✔ accounts for that program behave equally, ✗ represents programs behave unequally, ? represents that results are unknown, O/M represents out of memory.

The table shows the runtime and the behavior of the programs which are instrumented and are sliced. See Table 4.6.
Table 4.2: Runtime of the nonfunctional equivalence checker and the behavior of the programs that are instrumented and sliced

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Instrumented version</th>
<th>Sliced version</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \geq )</td>
<td>( = )</td>
</tr>
<tr>
<td>Integers</td>
<td>TB</td>
<td>TB</td>
</tr>
<tr>
<td>Simpleloop</td>
<td>10.9</td>
<td>✔</td>
</tr>
<tr>
<td>calculator</td>
<td>0.6</td>
<td>✔</td>
</tr>
<tr>
<td>Reverse</td>
<td>4.3</td>
<td>✔</td>
</tr>
<tr>
<td>Digit10</td>
<td>2.0</td>
<td>✗</td>
</tr>
<tr>
<td>Fibo</td>
<td>T/O</td>
<td>0.8</td>
</tr>
<tr>
<td>Whiled</td>
<td>1.5</td>
<td>✗</td>
</tr>
<tr>
<td>Sumofdigits</td>
<td>T/O</td>
<td>1.2</td>
</tr>
<tr>
<td>Nestedwhile</td>
<td>2.6</td>
<td>✗</td>
</tr>
<tr>
<td>Battle</td>
<td>5.2</td>
<td>✗</td>
</tr>
<tr>
<td>Heaps and pointers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insertion sort</td>
<td>2.8</td>
<td>✔</td>
</tr>
<tr>
<td>Bubble sort</td>
<td>15.3</td>
<td>✗</td>
</tr>
<tr>
<td>Selection sort</td>
<td>10.5</td>
<td>?</td>
</tr>
<tr>
<td>Memcopy_a</td>
<td>2.8</td>
<td>?</td>
</tr>
<tr>
<td>Memcopy_b</td>
<td>T/O</td>
<td>1.9</td>
</tr>
<tr>
<td>Clearstr</td>
<td>T/O</td>
<td>4.5</td>
</tr>
<tr>
<td>Propagate</td>
<td>5.9</td>
<td>✔</td>
</tr>
<tr>
<td>Swaparray</td>
<td>T/O</td>
<td>7.4</td>
</tr>
<tr>
<td>Reversearray</td>
<td>T/O</td>
<td>4.7</td>
</tr>
<tr>
<td>Prefetched</td>
<td>T/O</td>
<td>T/O</td>
</tr>
<tr>
<td>Loopfusion</td>
<td>21.8</td>
<td>?</td>
</tr>
<tr>
<td>Binarysearch</td>
<td>T/O</td>
<td>13.0</td>
</tr>
<tr>
<td>Linearsearch</td>
<td>T/O</td>
<td>3.0</td>
</tr>
<tr>
<td>Tunnelloopfusion</td>
<td>T/O</td>
<td>T/O</td>
</tr>
<tr>
<td>Looppeel</td>
<td>T/O</td>
<td>3.7</td>
</tr>
</tbody>
</table>

Recursion

<table>
<thead>
<tr>
<th>Limit</th>
<th>( \geq )</th>
<th>( = )</th>
<th>( &lt; )</th>
<th>( \geq )</th>
<th>( = )</th>
<th>( &lt; )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.6</td>
<td>✗</td>
<td>0.5</td>
<td>✗</td>
<td>0.6</td>
<td>✔</td>
<td>0.5</td>
</tr>
</tbody>
</table>
Chapter 5

Discussion and Conclusions

A full range of C programs was used and instrumented to observe the nonfunctional property of a program, in particular, we have analyzed the abstract execution time. An existing automated tool called REVE is used to measure the execution time(abstract) of the instrumented programs.

Each statement in a program will consume some amount of time(cycles) to execute. To measure the abstract execution time, we use a variable called cycle but renamed it as $C$ for easy understanding and implementation during the experiments. We have used the variable $C$ for instrumentation to address the execution time(abstract). A tool called REVE can check the functional properties of a program, in particular, the program equivalence(partial). We used instrumentation( the process of measuring the number of cycles to address execution time(abstract) consumed for each statement in the program) with the nonfunctional properties of a program, in particular, the abstract execution time of a program and translated(mapped) them into functional properties of a program. C programs that are written using a for loop are rewritten using a while loop since the REVE tool cannot handle the for loop.

The time referred here is not the absolute execution time of the program but the relative execution time of a program. It is quite difficult to measure the absolute execution time in our experimentation. Instrumentation is used to address the abstract execution time of a program based on the value of the variable $C$. After the instrumentation if the count value of the variable $C$ in the programs is equal we can conclude that both the programs need the same execution time. If the count value of the variable $C$ is more or less, then we can say that one program needs more or less time to execute than the other program.

When addressing the nonfunctional property in particular abstract execution time, most of the work is done manually. We have incremented the variable $C$ for every statement according to the requirement. After the instrumentation of the programs,
most of the programs showed a behavior of equal and a few demonstrated a behavior of unequal with some cases as the timeout.

After the process of instrumentation we use slicing (to comment those statements that do not affect the loop operation) to observe if there is a significant change in the execution time of the program. We see a broad range of results like equal, unequal, unknown and timeout with a different runtime on the tool. See table 4.6.

Especially programs that have operations such as multiplication and division had a result of timeout or unknown because the tool is amateur and cannot handle it. The slicing of programs has shown a mixed response where result show a less significant time to compute when compared to instrumented version of the programs.

Many observations can be made from the table, see Table 4.6. The tool needs more time to examine the programs that have arrays. There is a significant difference in the run-time of the tool between the instrumented versions and the sliced versions. Moreover, partial equivalence has been checked for all the inputs.

5.1 Future work

An automatic tool can be developed which can check both program equivalence and can perform instrumentation (checking the abstract execution time) with an automated approach. The program can also be modified to consider the cache and can handle the memory consumption as well. The REVE tool can only handle data types of integer and pointers to integers.

The REVE tool can be modified to consider programs of other data types such as float, character also. Many things in C language which are not supported by this tool can be used in the program like break and continue. There can also be a possibility of using the return statement in multiple places also. As of now the return statement is used only at the end, or the return statement must be the last statement of the program.
Bibliography


Chapter 6

Appendix

This section explains the list of programs that are considered to behave equally in which we have all the three versions namely equal, instrumented and sliced versions are available.

Here is the link for all the files that have been instrumented and sliced. https://github.com/mohitbv5244/mycode

```c
int digit10_1(int n) {
    int result = 1;
    n = n / 10;

    while (n > 0) {
        result++;
        n = n / 10;
    }

    return result;
}
```

Figure 6.1: (a) Original version of digits10 Program_1
int digit10_2(int n) {
    int result = 1;
    int b = 1;
    int retval = -1;

    while (! (b == 0)) {
        if (n<10) {
            retval = result ; b = 0; }
        else
            if (n<100) {
                retval = result+1; b = 0; }
            else
                if (n<1000) {
                    retval = result+2; b = 0; }
                else
                    if (n<10000) {
                        retval = result+3; b = 0; }
                    else {
                        n = n / 10000;
                        result = result + 4;
                    }
        { 
            return retval;
        }
    }
    return result;
}

Figure 6.2: (b) Original version of digits10 Program_2

int digit10_1(int n) {
    int result = 1;
    int C = 0;
    int cost_\> = 2;
    int cost_\\ = 4;
    int cost_\++ = 1;

    n = n / 10;
    C = C + cost_\\;
    while (n > 0) {
        C = C + cost_\>;
        result++; C = C + cost_\++;
        n = n / 10;
        C = C + cost_\\;
    }
    C = C + cost_\>
    return C;
}

Figure 6.3: (a) Instrumented version of digits10 Program_1
```c
/* rel_out \result <= \other \result @*/

int digit10_2(int n) {
    int result = 1;
    int b = 1;
    int retval = -1;
    int C = 0;
    int cost_\> = 2;
    int cost_\> = 4;
    int cost_++ = 1;

    while (! (b == 0)) {
        C = C + 2 * cost_\>;
        if (n<10) { 
            C = C + cost_\>;
            retval = result;
            b = 0; }
        else 
            if (n<100) { 
                C = C + cost_\>;
                retval = result+1;
                C = C + pluscost;
                b = 0; }
            else 
                if (n<1000) { 
                    C = C + cost_\>;
                    retval = result+2;
                    C = C + pluscost;
                    b = 0; }
                else 
                    if (n<10000) { 
                        C = C + cost_\>;
                        C = C + pluscost;
                        b = 0; }
                    else { 
                        C = C + cost_\>;
                        n = n / 10000;
                        C = C + cost_\>;
                        result = result + 4;
                        C = C + cost_++;
                    }
        C = C + cost_\>;
    }
    C = C + cost_\> + cost_\>;
    return C;
}
```

Figure 6.4: (b) Instrumented version of digits10 Program_2
```c
int digit10(int n) {
    int result = 1;
    int C = 0;
    int cost_\textgreater = 2;
    int cost_\textless\textgreater = 4;
    int cost_++ = 1;
    n = n / 10;
    C = C + cost_\textless\textgreater;
    while (n > 0) {
        // result++;
        C = C + cost_++;
        n = n / 10;
        C = C + cost_\textless\textgreater;
    }
    C = C + cost_\textgreater;
    return C;
}
```

Figure 6.5: (a) Sliced version of digits10 Program_1
```c
/*@ rel_out */
result = \result < \other \result */

int digit10,2 (int n) {
  int result = 1;
  int b = 1;
  int retval = -1;
  int C = 0;
  int cost_\_\_\_ = 2;
  int cost_\_\_\_ = 4;
  int cost_++ = 1;

  while (! (b == 0)) {
    C = C + 2 * cost_\_\_;
    if (n<10) { C = C + cost_\_;
      // retina = result; 
      b = 0; }
    else
      if (n<100) { C = C + cost_\_;
        // retina = result+1;
        C = C + cost_++;
        b = 0; } 
    else
      if (n<1000) { C = C + cost_\_;
        // retina = result+2;
        C = C + cost_++;
        b = 0; } 
    else
      if (n<10000) { C = C + cost_\_;
        // retina = result+3;
        C = C + cost_++;
        b = 0; } 
    else 
      C = C + cost_\_;
      // n = n / 10000;
      C = C + cost_\_\_;
      // retina = result + 4;
      C = C + cost_++;
    }
    C = C + cost_\_;
  }
  \result = result + 4;
  C = C + cost_\_ + cost_\_;
  return C;
}
```

Figure 6.6: (b) Sliced version of digits10 Program,2
void selsort_1(int *a, int n) {
    int j = 0;
    int i = 0;
    int t = 0;
    while(i < n) {
        j = i;
        while(j < n) {
            if (a[j] <= a[i]) {
                t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
            j ++;
        }
        i ++;
    }
}

Figure 6.7: (a) Original version of Selection sort Program_1

void selsort_2(int *a, int n) {
    int i = 0;
    int j = 0;
    int t = 0;
    int position = 0;
    while(i < (n - 1)) {
        position = i;
        j = n - 1;
        while(j < n) {
            if (a[position] > a[j]){
                position = j;
            }
            j ++;
        }
        if (position != i){
            t = a[i];
            a[i] = a[position];
            a[position] = t;
        }
        i ++;
    }
}

Figure 6.8: (b) Original version of Selection sort Program_2
```c
int selsort_1(int *a, int n) {
    int j = 0;
    int i = 0;
    int t = 0;
    int cost_1 = 2;
    int cost_2 = 2;
    int cost_3 = 2;
    int cost_4 = 1;
    while (i < n) {
        C = C + cost_1;
        j = i;
        while (j < n) {
            C = C + cost_2;
            if (a[j] < a[i]) {
                t = a[i];
                a[i] = a[j];
                C = C + cost_3;
                a[j] = t;
            }
            C = C + cost_1 + cost_2;
            j++;
        }
        C = C + cost_1;
        i++;
    }
    C = C + cost_4;
    return C;
}
```

Figure 6.9: (a) Instrumented version of Selection sort Program_1
```c
/*@ rel_out \result <= \other \result @*/

int selsort_2 (int *a, int n) {
    int i = 0;
    int j = 0;
    int t = 0;
    int position = 0;
    int C = 0;
    int cost_> = 2;
    int cost_== = 2;
    int cost_+= 1;
    int cost_-= = 1;
    int cost_R = 2;

    while (i < (n - 1)) {
        C = C + cost_>;

        position = i;
        j = n - 1;
        C = C - cost_==;
        while (j < n) {
            C = C + cost_>;
            if (a[position] > a[j]) {
                C = C + cost_R + cost_>;
                position = j;
            }
            C = C + cost_R + cost_>;
            j++;
            C = C + cost_+=;
        }
        C = C + cost_>;

        if (position != i) {
            C = C + cost_>;
            t = a[i];
            a[i] = a[position];
            a[position] = t;
        }
        C = C + cost_>;
        i++;
        C = C + cost_+=;
    }

    C = C + cost_>;
    return C;
}
```

Figure 6.10: (b) Instrumented version of Selection sort Program_2
int selsort_1(int a[], int n) {
    int j = 0;
    int i = 0;
    int t = 0;
    int cost_R = 2;
    int cost_> = 2;
    int cost_A = 2;
    int cost_R = 1;
    while (i < n) {
        C = C + cost_>;
        j = i;
        while (j < n) {
            C = C + cost_>;
            if (a[j] <= a[i]) {
                C = C + cost_R + cost_>;
                t = a[i];
                a[i] = a[j];
                C = C + cost_=;
                a[j] = t;
            }
            C = C + cost_R + cost_>;
            j++;
            C = C + cost_R;
        }
        C = C + cost_>;
        i++;
        C = C + cost_R;
    }
    C = C + cost_>;
    return C;
}
/* @ rel_out \result <= \other \result @*/

int sel_sort_2(int *a, int n) {
    int i = 0;
    int j = 0;
    int t = 0;
    int position = 0;
    int C = 0;
    int \text{cost}_> = 2;
    int \text{cost}_== = 2;
    int \text{cost}_++ = 1;
    int \text{cost}_-- = 1;
    int \text{cost}_R = 2;

    while (i < (n − 1)) {
        C = C + \text{cost}_-- + \text{cost}_>;
        position = i;
        j = n − 1;
        C = C - \text{cost}_--;
        while (j < n) {
            C = C + \text{cost}_>;
            if (a[position] > a[j]) {
                C = C + \text{cost}_R + \text{cost}_>;
                position = j;
            }
            C = C + \text{cost}_R + \text{cost}_>;
            j++;
            C = C + \text{cost}_++;
        }
        C = C + \text{cost}_>;
        if (position != i) {
            C = C + \text{cost}_>;
            t = a[i];
            a[i] = a[position];
            a[position] = t;
        }
        C = C + \text{cost}_>;
        i++;
        C = C + \text{cost}_++;
    }
    C = C + \text{cost}_>;
    return C;
}

Figure 6.12: (b) Sliced version of Selection sort Program_2
```c
void clearstr1(int *a, int n) {
    int i = 0;
    while(a[i] != 0) {
        a[i] = 0;
        i++;
    }
}
```

Figure 6.13: (a) Original version of Clear string Program_1

```c
void clearstr2(int *a, int n) {
    while(*a != 0) {
        *a = 0;
        a++;
    }
}
```

Figure 6.14: (b) Original version of Clear string Program_2

```c
int clearstr1(int *a) {
    int i = 0;
    int C = 0;
    int cost_> = 2;
    int cost_++ = 1;
    int cost_R = 2;
    int cost_= = 2;
    while(a[i] != 0) {
        C = C + cost_R + cost_>;
        a[i] = 0;
        C = C + cost_= = ;
        i++;
        C = C + cost_++ ;
    }
    return C;
}
```

Figure 6.15: (a) Instrumented version of Clear string Program_1
/* rel_out */

```c
int clearstr_2(int *a) {
    int *a0 = a;
    int C = 0;
    int cost_> = 2;
    int cost_== = 2;
    int cost_++ = 1;
    int cost_R = 2;

    while (*a != 0) {
        C = C + cost_R + cost_>
        +a = 0;
        C = C + cost_==;
        a++;
        C = C + cost_++;
    }
    C = C + cost_R + cost_>
    return C;
}
```

Figure 6.16: (b) Instrumented version of Clear string Program_2

```c
int clearstr_1(int *a) {
    int i = 0;
    int C = 0;
    int cost_> = 2;
    int cost_== = 2;
    int cost_++ = 1;
    int cost_R = 2;

    while (a[i] != 0) {
        C = C + cost_R + cost_>
        a[i] = 0;
        C = C + cost_==;
        i++;
        C = C + cost_++;
    }
    C = C + cost_R + cost_>
    return C;
}
```

Figure 6.17: (a) Sliced version of Clear string Program_1
```c
/* @ rel_out \result <= \other \result */

int clearstr_2(int *a) {
    int *a0 = a;
    int C = 0;
    int cost_>= 2;
    int cost_== = 2;
    int cost_++ = 1;
    int cost_R = 2;

    while(*a != 0) {
        C = C + cost_R + cost_>
        *a = 0;
        C = C + cost_==;
        a++;
        C = C + cost_++;
    }
    C = C + cost_R + cost_>
    return C;
}
```

Figure 6.18: (b) Sliced version of Clear string Program_2
```c
int doublewhile_1(int n) {
    int i = 1;
    int x = 1;
    while (i <= n) {
        x = x * 5;
        i ++;
    }
    i = 0;
    while (i <= n) {
        x = x + i;
        i ++;
    }
    return x;
}
```

Figure 6.19: (a) Original version of Barthe Program_1

```c
int doublewhile_2(int n) {
    int i = 1;
    int x = 1;
    while (i <= n) {
        x = x * 5;
        i ++;
    }
    i = 1;
    while (i <= n) {
        x = x + i;
        i ++;
    }
    return x;
}
```

Figure 6.20: (b) Original version of Barthe Program_2

59
```c
int doublewhile_1(int n) {
    int i = 1;
    int x = 1;
    int C = 0;
    int cost_++ = 1;
    int cost_+ = 3;
    int cost_> = 2;

    while (i < n) {
        C = C + cost_>;
        x = x * 5;
        C = C + cost_+;
        i++;
        C = C + cost_++;
    }
    C = C + cost_>;
    i = 0;

    while (i <= n) {
        C = C + cost_>;
        x = x + i;
        C = C + cost_+;
        i++;
        C = C + cost_++;
    }
    C = C + cost_>;
    return C;
}
```

Figure 6.21: (a) Instrumented version of Barthe Program_1
/* @ rel_out \
result <= \other \result */

int doublewhile_2 (int n) {
    int i = 1;
    int x = 1;
    int C = 0;
    int cost_++ = 1;
    int cost_*= 3;
    int cost_>= 2;

    while (i <= n) {
        C = C + cost_>=;
        x = x * 5;
        C = C + cost_*;
        i++;
        C = C + cost_++;
    }
    C = C + cost_>

    i = 1;

    while (i <= n) {
        C = C + cost_>
        x = x + i;
        C = C + cost_++;
        i++;
        C = C + cost_++;
    }
    C = C + cost_>

    return C;
}

Figure 6.22: (b) Instrumented version of Barthe Program_2
```c
int dooublewhile(int n) {
    int i = 1;
    int x = 1;
    int C = 0;
    int cost_++ = 1;
    int cost_ += 3;
    int cost_++ = 2;

    while (i <= n) {
        C = C + cost_;;
        //x = x * 5;
        C = C + cost_+;
        i++;
        C = C + cost_++;
    }
    C = C + cost_;;

    i = 0;

    while (i <= n) {
        C = C + cost_;;
        //x = x + i;
        C = C + cost_+;
        i++;
        C = C + cost_++;
    }
    C = C + cost_;;

    return C;
}
```

Figure 6.23: (a) Sliced version of Barthe Program_1
```c
/*@ rel_out \result <= \other \result @*/

int doublewhile_2(int n) {
    int i = 1;
    int x = 1;
    int C = 0;
    int cost_++ = 1;
    int cost_++ = 3;
    int cost_++ = 2;

    while (i <= n) {
        C = C + cost_++;
        // x = x * 5;
        C = C + cost_++;
        i++;   
        C = C + cost_++;
    }
    C = C + cost_++;

    i = 1;

    while (i <= n) {
        C = C + cost_++;
        // x = x + i;
        C = C + cost_++;
        i++;   
        C = C + cost_++;
    }
    C = C + cost_++;

    return C;
}
```

Figure 6.24: (b) Sliced version of Barthe Program_2
```c
int binarysearch_1(int *a, int search, int n) {
    int first = 0;
    int last = n - 1;
    int middle = (first + last) / 2;
    int found = 0;
    while (first <= last) {
        middle = (first + last) / 2;
        if (a[middle] < search) {
            first = middle + 1;
        } else if (a[middle] == search) {
            return middle;
        } else if (a[middle] > search) {
            last = middle - 1;
        }
    }
    return -1;
}
```

Figure 6.25: (a) Original version of Binary search Program_1

```c
int binarysearch_2(int *a, int search, int n) {
    int first = 0;
    int last = n - 1;
    int middle = 0;
    int found = 0;
    while (first <= last) {
        middle = first + (last - first) / 2;
        if (a[middle] < search) {
            first = middle + 1;
        } else if (a[middle] == search) {
            found = 1;
            return middle;
        } else if (a[middle] > search) {
            last = middle - 1;
        }
    }
    return -1;
}
```

Figure 6.26: (b) Original version of Binary search Program_2
```c
int binarysearch(int *a, int search, int n) {
    int first = 0;
    int last = n - 1;
    int middle = (first + last)/2;
    int found = 0;
    int C = 0;
    int cost_> = 2;
    int cost_++ = 1;
    int cost_- = 1;
    int cost_\ = 4;
    int cost_R = 2;
    while (first <= last) {
        C = C + cost_>
        middle = (first + last) / 2;
        C = C + cost_\ + cost_++;
        if (a[middle] < search) {
            C = C + cost_R + cost_>
            first = middle + 1;
            C = C + cost_++;
        }
        else if (a[middle] == search) {
            C = C + cost_R + cost_>
            found = 1;
        }
        else if (a[middle] > search) {
            last = middle - 1;
            C = C + minuscost;
        }
        C = C + cost_R + cost_>
    }
    return C;
}
```

Figure 6.27: (a) Instrumented version of Binary search Program_1
/* rel_out */

int binarysearch_2(int *a, int search, int n) {
    int first = 0;
    int last = n - 1;
    int middle = 0;
    int found = 0;

    int C = 0;
    int cost_> = 2;
    int cost_++ = 1;
    int cost_-- = 1;
    int cost_\_ = 4;
    int cost_R = 2;

    while (first <= last) {
        middle = first + (last - first) / 2;
        C = C + cost_\_ + cost_-- + cost_++;
        if (a[middle] < search){
            first = middle + 1;
            C = C + cost_++;
        } else if (a[middle] == search){
            found = 1;
            C = C + cost_R + cost_>
        } else if (a[middle] > search){
            last = middle - 1;
            C = C + cost_--;
        } C = C + cost_R;
    }
    return C;
}

Figure 6.28: (b) Instrumented version of Binary search Program_2

66
```c
int binarysearch_1(int *a, int search, int n) {
    int first = 0;
    int last = n - 1;
    int middle = (first + last) / 2;
    int found = 0;
    int C = 0;
    int cost_> = 2;
    int cost_++ = 1;
    int cost_- = 1;
    int cost_\ = 4;
    int cost_R = 2;

    while (first <= last) {
        middle = (first + last) / 2;
        C = C + cost_>;
        C = C + cost_\ + cost_++;
        if (a[middle] < search) {
            first = middle + 1;
            C = C + cost_++;
        } else if (a[middle] == search) {
            // found = 1;
            C = C + cost_R + cost_>;
        } else if (a[middle] > search) {
            last = middle - 1;
            C = C + minuscost;
        }
        C = C + cost_R + cost_>;
    }
    return C;
}
```

Figure 6.29: (a) Sliced version of Binary search Program_1
Figure 6.30: (b) Sliced version of Binary search Program_2