



**Visualization of data-flow in the context of
C/C++ code**

Master Thesis

Dept. of Computer Science
Software Engineering

In cooperation with
Vector Informatik GmbH

Submitted by:
Amalnath Joseph Nirmalan
Automotive Software Engineering
Matriculation Number: XXXXXXXXXX

Under the Guidance of:

Prof. Dr. -Ing Janet Siegmund
Department of Computer Science
Software Engineering

Timo Vanoni
PES3.1.1
Vector Informatik GmbH

Acknowledgement

Firstly, I would like to thank my thesis advisor Prof. Dr. Janet Siegmund, Dept. of Computer Science, TU Chemnitz, for approving my thesis topic and accept to pursue it under the Professorship of Software Engineering. I want to extend my thanks to Ms. Arooba Aqeel for the support and guidance provided during the thesis and for the feedback while writing the thesis report. She was always reachable, listened to my queries, got a quick response, and encouraged open discussion.

I would like to express my sincere gratitude and thanks to my supervisor at Vector Informatik, Mr. Timo Vanoni, for the continuous support of my Master thesis; he was always there to listen to my queries and ideas. I need to appreciate how he gives me advice and suggestions, resulting in the betterment of accomplishing the task. I am so grateful for his patience, motivation, and support in overcoming numerous hurdles that I have been facing. Also, I would like to specially mention the moral support he had given me during the period after the office's closure due to Covid-19. I would also like to thank my manager at Vector Informatik, Mr. Markus Schwarz, for giving me a great opportunity to join their team and for all the support he had offered. Even during his busy schedule, he finds time to discuss the development of thesis work, my comfort, and the feedback that helped me progress in my work. Finally, I would like to thank all my team members (PES 3.1.1) for the support and constructive feedback they had given to complete the thesis work; it helped me improvise as an individual and learned a lot. Thank you, Vector Informatik GmbH, for giving me this opportunity, and I will always be thankful for all the experiences and knowledge gained.

I am grateful to God, my parents, for the immense support and affection they had shown throughout the entire period. Finally, my sincere thanks go to my friends who stood by me during every phase, for the advices, encouragement, and support.

Abstract

Vector Code Analyzer (VCA) is a static code analysis tool that analyses the source code after the successful compilation, and the findings will be reported in text format. However, the findings reported by VCA are hard to comprehend by the user if the necessary pieces of information are not visualized. This thesis aims to provide a concept for the visualization of data flow so that the users concerned with the static code analysis can evaluate the result effectively and trace the reasons for findings straight away.

Data flow analysis is a technique used to analyze static source code for potential vulnerabilities like null pointer issues, divide by zero error, buffer overflow. With the help of data flow analysis, it is possible to obtain the possible set of values calculated at various points in a program, which helps understand the whole program quite efficiently.

There are several automated tools for static data flow analysis, but most of the tools have a limitation in data flow visualization. Mostly the existing tools will provide only the details of errors with line numbers, and then users have to analyze to find out the actual cause manually. It is a tedious task to trace the error and understand the code if there is no visualization method, depending on the number of code lines.

This thesis proposes a concept to solve the existing limitation in data-flow analysis and helps users understand the source code and the findings made by the vector code analyzer without any difficulty by implementing a simple and effective means of data flow visualization by a click and hovering of variables.

Keywords: Vector Code Analyzer, Static Analyzer, Data flow analysis, Visualization

Contents

Contents	4
List of Figures	6
List of Abbreviations	8
1 Introduction	9
1.1 Motivation	10
1.2 Research Questions	10
1.3 Thesis Outline	11
2 Code Analysis	12
2.1 Dynamic Code Analysis	12
2.2 Static Code Analysis	13
2.2.1 Common Defects	14
2.2.2 Static Analysis Techniques	18
3 Existing tools	22
3.1 Astrée	22
3.2 HelixQAC	24
3.3 Parasoft	25
4 Vector Code Analyzer	28
4.1 VCA Framework	28
4.2 Static Single Assignment Form	30
4.3 Abstract Interpretation	32
4.3.1 Semantics	33
4.3.2 Collecting Semantics	34
4.3.3 Abstract Semantics	34
4.3.4 Abstract Domain	34
4.3.5 Galois Connection	35
4.3.6 Widening/Narrowing	36
4.4 False Alarms	36
4.5 Sound and Unsound static analysis	37
4.6 Data flow Analysis	38
4.7 Clang/LLVM	41

CONTENTS

5 Existing state of Vector Code Analyzer	43
5.1 How Vector Code Analyzer works	43
5.1.1 Code optimization	44
5.1.2 Flow-insensitive pointer analysis	44
5.1.3 Approximate indirect callgraph	45
5.1.4 Calculate side effect for each function	46
5.1.5 Calculate def-use chains	47
5.1.6 Combined data flow analysis	47
5.2 Output of Vector Code Analyzer	48
5.3 Issues concerned with the users	49
6 Concept	51
6.1 Output window	51
6.1.1 Range analysis of variable	52
6.2 Visualization of straight-line code	54
6.3 Visualization of loops	56
6.4 Environment preparation	59
6.4.1 Vue.js	59
6.4.2 Monaco editor	60
7 Conclusion	61
8 Future Scope of Work	62
References	63

List of Figures

2.1	A typical static code analysis	13
2.2	The average cost of fixing defects depending on the time they have been made and detected	14
2.3	Example of Division by zero error	14
2.4	Example of Null Pointer Dereference	15
2.5	Example of memory leak	16
2.6	Example of buffer overflow	16
2.7	Example of Uninitialized variable	17
2.8	Type cast	18
2.9	Typical example of CFG with all basics blocks	19
2.10	Examples for general control flow graph	19
2.11	Basic Terminologies	20
2.12	Control and data flow analysis of a simple code	21
3.1	Performance of Astrée Code Analyzer	22
3.2	Result analysis by Astrée	23
3.3	Data flow analysis by Astrée	24
3.4	Result window of HelixQAC	25
3.5	Result window of Parasoft	26
3.6	Flow analysis by Parasoft	26
4.1	Code Analysis Framework	29
4.2	SSA Form of a simple straight-line code	30
4.3	Example code (left), Control flow graph (Middle), and the control flow graph in SSA Form (Right). ϕ function and versioned variables are shown (black background)	31
4.4	Steps of Abstract Interpretation	33
4.5	2D Graph representing Concrete Semantics	34
4.6	Galois Connection	35
4.7	False Alarm	37
4.8	Difference between a sound and unsound tool	38
4.9	Reaching Definition with two nodes	39
4.10	Reaching Definition with three nodes	39
4.11	Overview of LLVM compilation strategy	41
4.12	output of an error	42
5.1	Architecture of Vector Code Analyzer	43

LIST OF FIGURES

5.2	Analysis process of Vector Code Analyzer	44
5.3	Flow-insensitive pointer analysis	45
5.4	indirect callgraph	45
5.5	calculation of side effect	46
5.6	Output of VCA	48
6.1	Conceptual output window	51
6.2	Example for range propagation	53
6.3	Simple C code for swapping	54
6.4	Initial step of data flow analysis	55
6.5	Final step of data flow analysis	55
6.6	Example for folding	56
6.7	concept for loops	57
6.8	Code with decidability problem	58
6.9	Concept for the decidability problem	59

1 Introduction

Static code analysis or source code analysis analyzes the code without the code's actual execution. It is similar to White-box testing (tests the internal structures of an application), usually performed as part of code review during the implementation phase of the Security development life cycle. The importance of doing the static analysis is that it is always better to check for defects and fix errors early, which enables the user to significantly reduce the amount of downstream work added to the project and is a surefire method to avoid multiple obstacles along the way. The state of software is becoming more and more complex as well as sophisticated. Today, for example, the average automobile may contain over 1,000 code executing Microcontroller Units (MCU) and as much as 100 million lines of code! [18]. This enormous electronic surface area, especially in safety-critical applications, such as automotive, medical devices, or avionics software, demands a rigorous engineering approach to software to approach defect-free code. Failing to check with at least one static analysis tool significantly raises the risk of deploying or releasing the applications with defects, it can lead to exploitable code that malicious hackers can use to crash the system, expose sensitive data, and more. In the case of safety-critical software, the consequences of software vulnerabilities can be far more severe.

Several kinds of research are in progress to figure out a systematic method to find the defects and improvise the existing static analysis tools to tackle this situation. According to ISO 26262, it is necessary to check the code with an automated static code analysis tool [29]. The UK Defense Standard 00-55 requires using a Static Code Analysis tool on all safety-related software in defense equipment [25].

Static analyzers available today report many defects specifying their location of occurrence through line numbers in source code. A user of such tools must first go to the specified line number for each defect and then analyze the program manually to identify the root cause for each defect, which may be nontrivial. Hence, there is a need to provide an effective analysis aid to developers, facilitating easy identification of root causes of defects identified statically and understanding the data flow in general. This is where the importance of data flow visualization comes in. With the visualization of data flow, the complexities mentioned above can be minimized, and it eases the user to understand the findings made by the Vector Code Analyzer, which in turn results in reducing the time for understanding the code and finding the root cause for defects.

1.1 Motivation

As the coding becomes highly complex and complicated, analysis tools are also in great demand. It is a must to analyze the source code and verify it is error-free if it deals with critical scenarios, and any omission or unnoticed error might end up causing danger to human life. Some of the significant issues that occurred due to flawed static analysis:

- The recent recall of vehicles by major automotive manufacturers due to software errors [8].
- The explosion of Ariane 5 due to the software overflow error [15].
- The halting problem occurred in USS Yorktown due to a divide-by-zero error in a database application that propagated throughout the ship's control systems [33].

These are some of the incidents to highlight the importance of static analysis. This is why data flow and visualization are going to play a vital role. Data flow analysis is an essential technique for carrying out useful static analysis. It is crucial for understanding the code because it is impossible to analyze the code effectively without adequate information regarding how the data flow occurs within a particular code segment. However, no effective method is provided by any of the static analysis tools available to understand the source code's data flow. It is an area that remains unnoticed, or there is much room for improvement. This lack of visualization makes the analysis of findings/results generated by the static analyzers more complex. This thesis aims to provide a concept for visualizing the data flow so that the user is concerned with analysis can understand the code much better, thereby reducing the risk of errors going undetected or neglecting errors. The significance of data flow visualization and its role in ease out the findings made by Vector Code Analyzer leads to pursue this Master Thesis.

1.2 Research Questions

1. What is the information necessary to understand the finding of a static code analysis tool?
2. How can this information be visualized in a way a user can comprehend?
3. How to present only the information necessary for a specific finding?
4. How can the user specify what he is interested in (e.g., which finding)?

1.3 Thesis Outline

- Chapter 1: Introduction - The motivation of the thesis is explained with a basic introduction to the importance of DFA, and its visualization, the research questions are formulated.
- Chapter 2: Static Code Analysis - Briefly describes static code analysis, its techniques, and its significance.
- Chapter 3: Explains the existing static code analysis tools, analysis of the result, and existing tools' limitations.
- Chapter 4: Vector Code Analyzer - This section explains the static analysis tool VCA; its design and implementation.
- Chapter 5: Existing state of Vector Code Analyzer - This chapter will provide a better insight about the analysis and drawbacks of the output made by the analyzer.
- Chapter 6: Concept - This chapter focuses on the solutions for overcoming the existing situation.
- Chapter 7: Conclusion - A summary of the work and its relevance concerning the existing scenario.
- Chapter 8: Future scope of work - This chapter will discuss the improvements that can be made.

2 Code Analysis

Code analysis is the process of analyzing code to find whether the code is efficient, robust, safe, and to understand its correctness and its expected performance [5]. It mainly focuses on verifying code correctness and its optimization. Program correctness is intended to analyze whether the code/program is working as it is meant to be, and optimization is done to improve the code performance by efficiently using resources. The analysis can be performed either during the execution of a program that is during run-time (Dynamic Code Analysis) or without running the code (Static Code Analysis).

2.1 Dynamic Code Analysis

As its name indicates, dynamic code analysis is an analysis that performs during the code's execution. It is necessary to identify and create test inputs for performing dynamic code analysis and create a test strategy to cover almost all possible paths. The source code has to be compiled into an executable file; otherwise, it is impossible to perform the analysis as it does not support the code having compilation or build errors. Some of the dynamic testing examples are the unit test, integration test, system test, and acceptance test.

There exist several dynamic code analysis utilities and tools intended for executing, provide findings and analysis. Many contemporary development environments already have dynamic analysis tools as one of its modules. Such is, for example, the Microsoft Visual Studio 2012's debugger and profiler [13].

Dynamic analysis is executed by passing a set of data to the input of the program is checked. That is why the efficiency of analysis depends directly on the input test data's quality and quantity. It will help identify resources consumed, the degree of code coverage with tests, other program metrics, program errors, and vulnerabilities in the program [13].

Dynamic testing allows users to make sure that the product works well or reveals errors showing that the program does not work. The second goal of the testing is a more productive one from the viewpoint of quality enhancement, as it does not allow the users to ignore the program's drawbacks. Even though no defects have been revealed during the testing, it does not necessarily mean there are no errors at all. Even 100 percentage code coverage with tests does not mean there are no code errors since dynamic testing cannot reveal some issues like logical errors. Code coverage, memory error detection, fault localization, invariant inference, security analysis, program slicing, and performance analysis is some kind of dynamic

analysis [14].

2.2 Static Code Analysis

Static code analysis uses several techniques that analyze the programs by examining the source code without the actual execution of code. The main advantage and benefit of static code analysis are that it helps identify the defects and problematic code constructs during the early stage of development (between coding and unit testing), which results in improving software quality and cost reduction [71]. Static

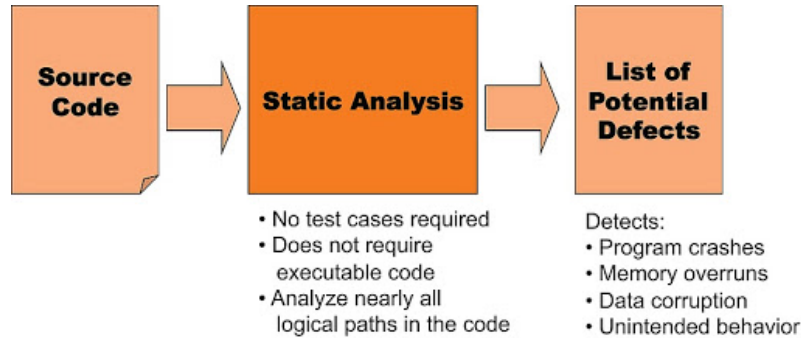


Figure 2.1: A typical static code analysis

code analysis plays an integral role in maintaining efficiency and code quality before unit testing, and it helps discover the hidden defects that might not be able to find during other verification and validation processes. Approximately 20 to 30 percent of all software will contain more than 50 percent defects, and static code analysis will reduce the post-release failures by more than 50 percent, which results in the reduction of development cost by not less than 30 percent [42]. Nowadays, as the software dealt with more safety-critical applications, with possible human lives depending on their uninterrupted functionality and the code complexity is getting high, static code analysis is being widely used, and because of this, there exist several static code analysis tools for prominent languages like C/C++, Python, C-sharp, and many others. How static code analysis differs from regular inspection and testing is, it works fully automated, and there is no need for elaborate test settings. Static code analysis identifies the program behavior by checking the program structure, elements or by taking the approximation of the program states.

	Time Detected				
Time Introduced	Requirements	Architecture	Construction	System Test	Post-Release
Requirements	1	3	5-10	10	10-100
Architecture	-	1	10	15	25-100
Construction	-	-	1	10	10-25

Figure 2.2: The average cost of fixing defects depending on the time they have been made and detected

[71]

2.2.1 Common Defects

As mentioned earlier, the static code analysis helps to discover some hidden defects. Some of the major defects that can be unearthed are:

- **Division by Zero**

It is a logical software error that occurred due to the division of a number by zero, and it leads to a runtime error because the value getting after division is undefined. When trying to divide a number by an integer zero on a processor of the x86 and x86/64 processor families, a hardware exception is generated (by vector 0); accordingly, a C/C++ program compiled into machine code will halt program execution or yields unpredictable computation results [12].

```
void DBZ(int a, int b)
{
    int c = a - b; // If a==b, then c will become zero
    b = b / c; // Possibly a divide by zero error
}
```

Figure 2.3: Example of Division by zero error

In the above code fragment, it shows the possible occurrence of division by zero error. If the parameters 'a' and 'b' have the same value, then 'c' might become zero, it will result from the division statement to zero, causing the program to crash. One of the significant incidents that occurred due to division by zero error is the

halting of USS Yorktown(CG-48). On September 21, 1997, a division by zero error in the "Remote Data Base Manager" aboard the ship brought down all the machines on the network, causing the ship's propulsion system to fail.

• Null Pointer Dereference

A null pointer is a pointer that indicates the memory address 0 (in most cases). According to the C standard, dereferencing a null pointer is undefined behavior, resulting in a program crash. It could continue working silently or be transformed into a software exception that can be caught by program code. There are, however, certain circumstances where this is not the case. For example, in x86 real mode, the address 0000:0000 is readable and usually writable, and dereferencing a pointer to that address is a perfectly reasonable but typically unwanted action that may lead to undefined, but non-crashing behavior in the application [22]. Most of the dereferencing cases will end with causing "segmentation fault" or "access violation", thereby results in the termination of the program by the operating system.

In the given example, it is clear that the pointer 'a' has a null value as it is initialized with NULL. When 'b' tries to read '*a', it points towards an invalid pointer and dereferencing occurs. The same will happen even in the case of writing a value.

```
int *a = NULL; // a is a null pointer
int b = *a;    // Program might crash: dereferencing null pointer
*a = 0;       // Program might crash: dereferencing null pointer
```

Figure 2.4: Example of Null Pointer Dereference

• Memory Leak

A memory leak occurs when the program allocates memory on the heap and forgets to de-allocate once it has finished using it. If there is a substantial amount of memory leakage occurs, then it is impossible to allocate memory further, and the program cannot continue execution.

The example in figure 2.5 shows that the memory leak occurs because the allocated memory will go out of the scope since the corresponding free statement is not provided.

```

void ML() /* Function with memory leak */
{
    int *ptr = (int *) malloc(sizeof(int));

    .....

    return; /* Return without freeing ptr*/
}

```

Figure 2.5: Example of memory leak

• Buffer Overflow or Underflow

A buffer is a temporary area for holding data. For any statically or dynamically allocated memory buffer, there is a limit for the maximum number of data elements it can store. A buffer will overflow when the data (meant to be written in the buffer) gets written past either before it's beginning or after it's ending. This way, the data will be stored in a portion of memory that does not belong to the actual program variable that references the buffer and corrupts the memory belongs to other variables or overwrite whatever data they were holding.

```

char buff[10];
buff[10] = 'a';

```

Figure 2.6: Example of buffer overflow

In the below-mentioned example, an array of size 10 bytes (range of the index is from 0 - 9) has been declared, and in the next line, tried to store the value 'a' in the index range 10, here buffer overflow will occur because data will get written out of the scope of the buffer, here beyond the right boundary of buffer. Underflow defects will occur similarly but in the other memory direction.

• (Arithmetic) Integer Overflow

An integer overflow occurs when the user attempts to store a value that is larger than the highest value an integer variable can hold. According to the C standard, it is undefined behavior, and anything might happen due to this. Integer overflows are the consequence of "wild" increments/multiplications, generally due to a lack of validation of the variables involved [54].

The explosion of the Ariane 5 rocket also happens because of an overflow (Integer overflow) defect. The on-board computer crashed 36.7 seconds after starting when trying to read the value of the horizon. One of the functions was to convert the speed of the 64-bit floating-point display to a 16-bit signed integer: $- + b_1 b_2 \dots b_{15}$. The corresponding number was greater than the permissible limit and generated an integer overflow, which results in the crash and loss of 500 million USD [61].

- **Uninitialized Variable**

An uninitialized variable is a variable, which is declared but is not provided with an initial value. In several programming languages, the initialization of the variable is optional. If the variable is not initialized at the time of declaration, then the variable will have a random value based on whatever was in the memory holding that variable. An uninitialized variable is similar to uninitialized memory and might cause an error during the program execution.

```
int UI(int n)
{
    int sub, i;

    for (i = 0; i < n; i++)
    {
        sub = sub - 15;
    }

    return sub;
}
```

Figure 2.7: Example of Uninitialized variable

The variable 'sub' in figure 2.7 is not initialized, and currently, it holds a garbage value. Sometimes the program might run, as the uninitialized variable will automatically set to value zero. However, in general, the output or the result of a function is unpredictable.

- **Inappropriate Cast**

Some coding languages support to cast one type of variable to another type, which means one data type to another (for example, Char to Int). Unfortunately, casting comes with several problems, and sometimes it may alter the variable's value (for example, bit truncation). Sometimes, the compiler might fail to warn about the problem, and it remains undetected. In the given code fragment, the function getchar will returns an integer value. However, the problem is that the value returned will have a range more than that of char variable 'c'. It will create severe problems, and the program will not behave as expected. For example, the value returned from getchar (256) will be considered as the value 0.


```

int char_io()
{
    char c;
    c = getchar(); // Returns an int value to c, which is a char
    return c;
}

```

Figure 2.8: Type cast

2.2.2 Static Analysis Techniques

To successfully trace the defect or vulnerabilities, there are various techniques to analyze static source code. All these techniques are integrated to get a better and efficient static code analysis. These techniques are often derived from compiler technologies. The primary static code analysis techniques are the following [25]:

- **Control flow analysis**

For understanding or doing static analysis, it is a must to know the program's control flow. Control flow analysis is a static code analysis technique that determines and identifies the code fragment's control flow. It will analyze all the possible execution paths inside a program or procedure; it will usually be flow-insensitive. A control flow graph (CFG) is used to express the control flow.

A control flow graph is the graphical representation of control flow; it denotes all paths that might be traversed through a program during its execution. It is essential for performing static analysis; it can be constructed directly from the program by scanning it for basic blocks; it is a linear piece of code without any jumps or jump targets. As a directed graph, the node in the graph will represent a basic block, and the edges represent the transfer of control or jumps in the control flow. A control flow graph consists of two designated blocks to denote the Entry block (it allows the control to enter into the control flow graph) and Exit block (Control flow terminates via exit block). Usually, there will be only one path leading from the entry block to the exit block.

In figure 2.9, basic blocks are indicated as BB and in the given code snippet contains four basic blocks.

2 Code Analysis

```
int cfg = 21;
while (cfg < 25)
{
    cfg++;
    printf("%d" ,a);
}
return 0;
```

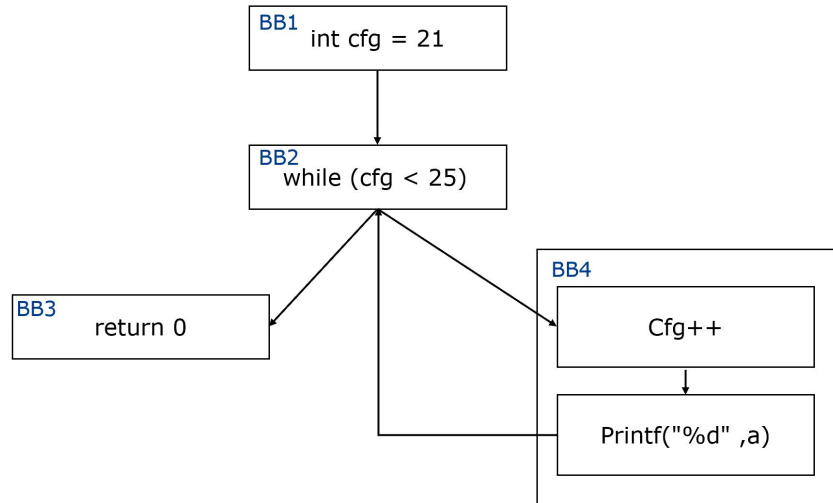


Figure 2.9: Typical example of CFG with all basics blocks

Some examples for general control flow graphs are:

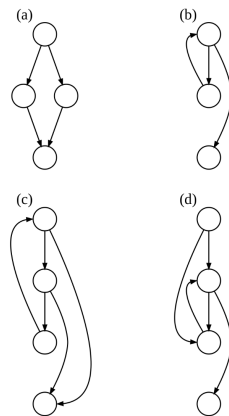


Figure 2.10: Examples for general control flow graph
[7]

- (a) denotes the if-then-else loop
- (b) denotes the while loop
- (c) denotes a loop with two exits
- (d) denotes a loop with two entry points

- **Data flow analysis**

Data flow analysis is a technique used for gathering information about the possible set of values calculated at various points in a program. It is usually performed on the program's control flow graph. Some of the categories of data flow analysis are flow-sensitive and path-sensitive analysis. The flow-sensitive analysis is done based on considering only the order of the control flow graph's statements. The path-sensitive analysis will also consider the conditional information at branch points and how they influence the possible values carried by a variable.

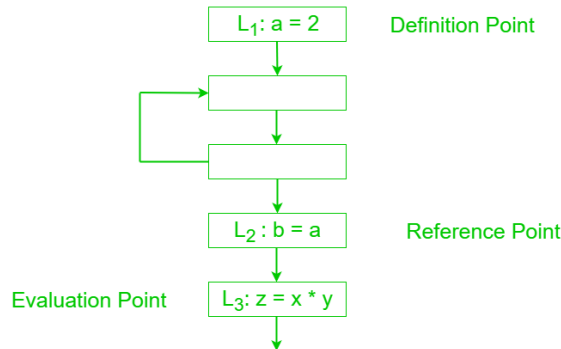


Figure 2.11: Basic Terminologies
[10]

Definition point: It will provide the definition

Reference point: It denotes a reference to data item

Evaluation point: This point contains the evaluation of expressions [10]

The figure given below contains the control and data flow analysis of a particular code snippet. Here in this, sometimes multiple values will flow into the same location. Consider the last print statement because it might take the value from the first print statement where x is one or from the final conditional statement where the value of x is 2. To understand how the actual data flow occurs, we need to know the control flow first. In figure 2.12, control flow is denoted using the black arrow, and data flow is denoted using red arrow edges. In the beginning, the value of x will be one, and it will not change until the conditional statement. After the conditional statement, the value changes to 2.

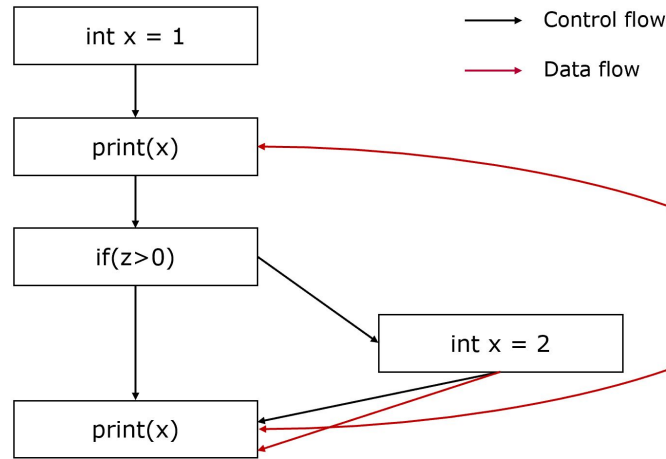


Figure 2.12: Control and data flow analysis of a simple code

- **Taint analysis**

The taint analysis is a popular method that consists of checking which variables can be modified by the user input. All user input can be dangerous if they are not properly checked. With this method, it is possible to check the registers and the user's memory areas when a crash occurs. If the tainted variable gets passed to a sink (vulnerable functions) without first being sanitized, it is flagged as a vulnerability [25].

- **Lexical analysis**

Lexical analysis converts source code syntax into tokens of information to abstract the source code and makes it easier to manipulate. It is the first phase of the compiler, known as the scanner. Some examples of a token are Type token (id, number), Punctuation token (void, return), Alphabetic token (Keyword) [25].

3 Existing tools

As mentioned earlier, there exist several tools for performing static code analysis. This section will discuss some of the tools which are comparable with VCA.

3.1 Astrée

Astrée stands for *Analyseur statique de logiciels temps-réel embarqués* (real-time embedded software static analyzer) [2]. It is a static code analyzer which is specially meant for analyzing the software written in C-programming language, and it analyzes and proves the absence of runtime errors and faulty behavior in safety-critical real-time synchronous C programs.[16, 49]

Astrée performs the static code analysis based on abstract interpretation (described

Nb of lines	70 000	226 000	400 000
Number of iterations	32	51	88
Memory	599 Mb	1.3 Gb	2.2 Gb
Time	46mn	3h57mn	11h48mn
False alarms	0	0	0

Figure 3.1: Performance of Astrée Code Analyzer
[50]

in the earlier section). While evaluating the analysis results, one thing can clearly understand that it performs well (see figure 3.1). C-code analysis complies with the ISO/IEC 9899:1999 C programming language standard and other safety standards like ISO 26262. It supports the C programs with loops, pointers, function calls, structures and arrays, integer and floating-point computation, and some extend of branching. However, certain C program characteristics are excluded from the scope analysis like backward branching, union types, dynamic memory allocation, unbounded recursive function calls, and the use of C libraries [50, 47]. It also includes a rule checker which helps to check the code compatibility with certain coding rules (MISRA, AUTOSAR, CWE, ISO/IEC, SEI CERT C) [16].

Some of the major characteristics of Astrée analyzer are:

- It is a sound program analyzer (never failed to point an error that can appear in some execution environment)

3 Existing tools

- Automatic (no end-user intervention is needed after parameterization)
- Highly efficient (any complex code analysis can be performed within a short period)
- Very precise (very few or complete eradication of false alarms)

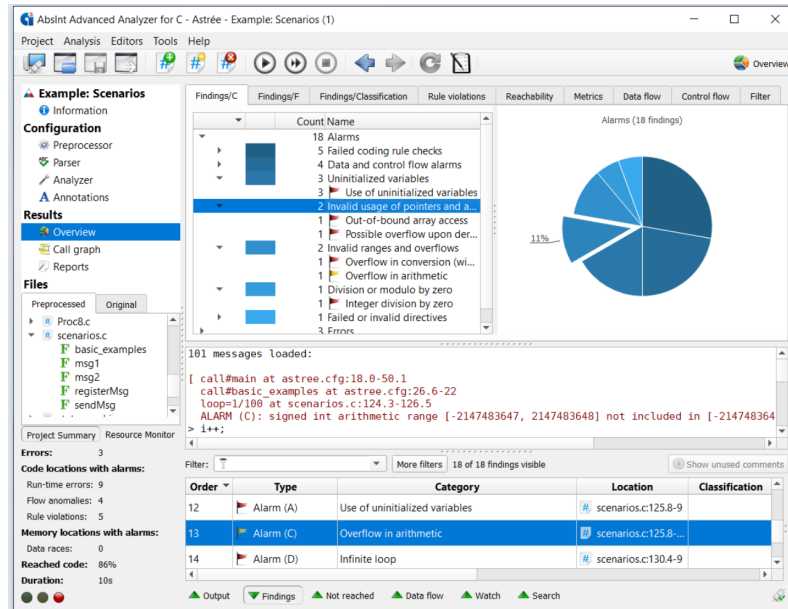


Figure 3.2: Result analysis by Astrée [16]

Figure 3.2 shows the results and findings made after the analysis. The analysis outputs a list of alarms (potential errors); these errors are reported along with their class and the location of occurrence. The occurrence or non-occurrence of errors can be easily understood, thanks to the user-friendly GUI. It is classified based on the colors (Red, Green, Yellow).

Red: Occurrence of at least one error

Yellow: Zero error, but at least one alarm of Class A (a potential run time error with an unpredictable result) or B (a potential data race)

Green and Yellow: Zero error, but at least one alarm of class C (a potential runtime error with a predictable result)

Green: No errors and alarms of class A, B, and C

It is possible to generate a report and also provide the code coverage information [27, 16].

3 Existing tools

Along with these, it will display all the data flow information collected by the analyzer. In figure 3.3 displays all accesses to global/static variables and provides information like whether it is shared or not, involved in a data race (occurs when two or more threads in a single process access the same memory location concurrently), the origin of the variable (from which function/process). The context menu supports tracing the corresponding code location of variables.

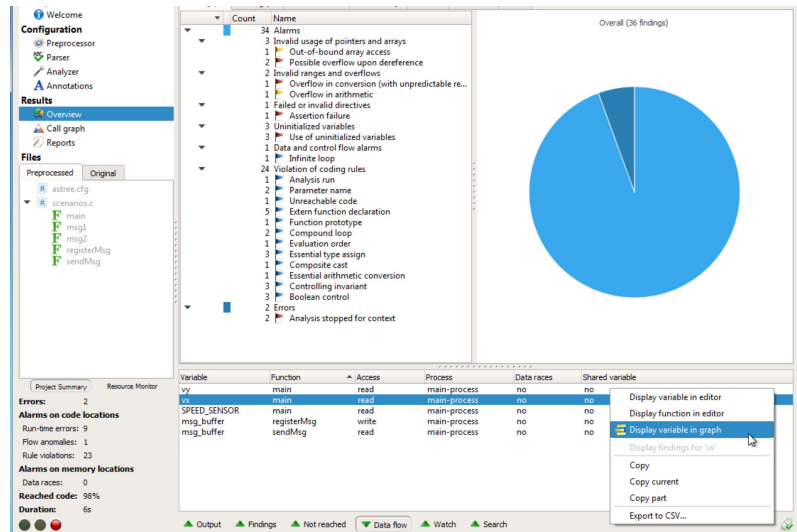


Figure 3.3: Data flow analysis by Astrée

3.2 HelixQAC

It is a static code analyzer from Perforce, automatically scans (based on C and C++ coding rules) codes to find out errors and abnormal behavior. It supports various coding and industry compliance standards, and then the report generated at the end will specify and help realize which parts of the code need improvements. It provides sufficient code anomalies coverage as it tracks the value of variables in the code as they would be at run time and prioritizes the defects based on their severity, which means it will identify those must-fix defects and provide detailed instructions to developers to fix the source code defect. The prioritization of defects is done by making use of specific filters, suppression, and baselines. Figure 3.4 shows the result window of HelixQAC; it will help understand the analysis result. It will display the errors, and it can be filtered based on their severity, the path of source files, and source language. It will also enable the user to comment on the results found by QAC and assign particular actions that have to be carried out. Like the Astrée Code Analyzer, it will also support generating the report and supports handling millions of code lines [26, 38].

At first glance, the result window seems sufficient, but as the complexity of the code increases, the user might find it difficult to analyze the result provided by QAC.

3 Existing tools

Because here in the result window provides the severity, cause, location, and cause of the error, but nothing to trace back the error's origin. Due to this, the user will be forced to analyze the code manually, and depending on the lines of code, complexity increases exponentially. This is one of the biggest hurdles faced by the users these days; even after getting so much information after the analysis, it will not be helpful if there is no way to identify the root cause of error and data flow of variable.

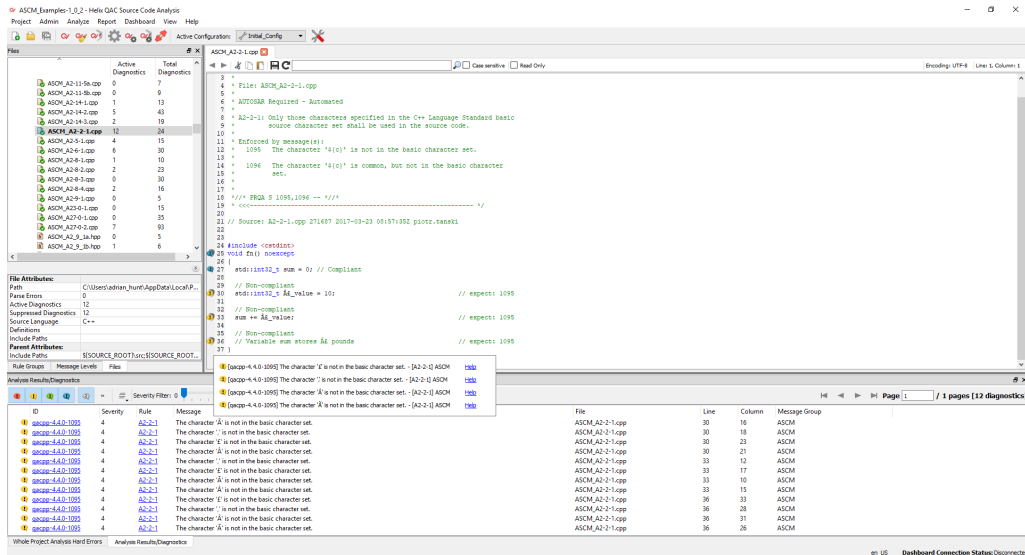


Figure 3.4: Result window of HelixQAC [26]

3.3 Parasoft

Another static code analyzer for analyzing the C/C++ code performs the analysis based on static analysis techniques (pattern-based analysis, data flow analysis, abstract interpretation, metrics, and more).

Pattern-based analysis detects constructs in the source code that are known to result in software defects based on programming standards, such as CWE and OWASP. It helps ensure that developers are following coding best practices, unit testing best practices, and the organization's development policy [1].

According to Parasoft, it comes with the largest number of checkers in the industry, resulting in a better evaluation of code. Like the other tools mentioned above, this tool also supports the industry standards and C/C++ programming standards. It supports performing analysis either in the IDE (Eclipse, Visual Studio, Visual Studio Code) or in the command-line interface (for automation/continuous integration scenarios). The result of the analysis can be generated and viewed either in the IDE or it can be viewed as HTML/PDF/XML reports. It also supports the users in managing testing results, prioritizing findings, suppressing unwanted findings, and

3 Existing tools

assigning findings to developers [3].

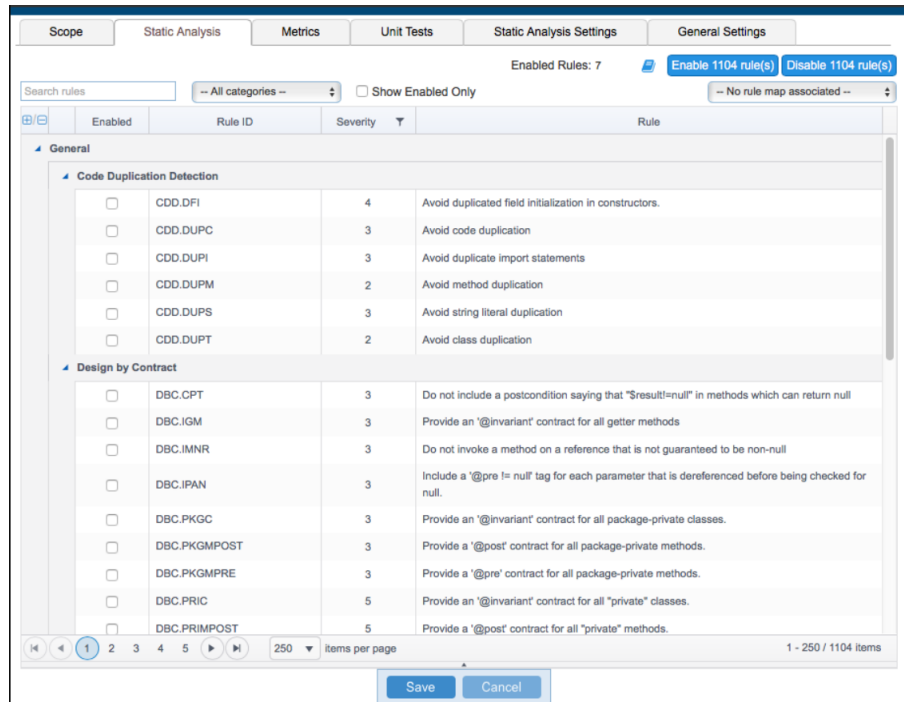


Figure 3.5: Result window of Parasoft [6]

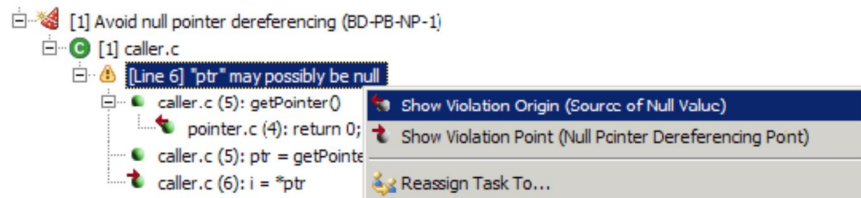


Figure 3.6: Flow analysis by Parasoft [17]

In the quality tasks view of Parasoft, violation/errors are represented as a hierarchical flow path, leading to identifying the problem. Flow path elements are represented with different icons, and each has a different meaning. For example, the red arrow-headed sphere indicates that the flow does not proceed as normal. Each element in the flow path has a tooltip, and while hovering over, it provides details like error description, path, variable causing error. It also provides the two most important pieces of information:

3 Existing tools

- Violation cause - This shows the source of the violation.
- Violation point - This is the point where flawed data has been used and results in an error.

Several other static code analyzers exist apart from those mentioned above, and all these will do the analysis and yield satisfactory results. However, these analyzers have limitations when it comes to the visualization of data flow. In all these three tools that are considered, each tool has its own limitations. In HelixQAC, it provides so much important information, but there is no effective mechanism to analyze the provided result. The tools Astrée and Parasoft come with a mechanism to understand the data flow and the results, but considering from user side, it has to be improved. These two tools provide necessary information about the data flow of variable and help to identify the source of the error.

With the existing static code analyzers, the user can perform the source code analysis, view the reports, and analyze it with various means (graph, chart, table) and help the user to understand whether the source code is error-free. The main problem is if the user wants to understand/analyze the code and trace the actual reason for the error, the scope of these tools is limited. Thus, even after performing the analysis, the user has to go through the code to get a better understanding of the code. It will become more time-consuming and complex depending on the number of lines of code. All static code analyzers mentioned above use data flow to perform the analysis, but there is no effective method for visualizing the data flow.

4 Vector Code Analyzer

Vector Code Analyzer is a static code analyzer based on Clang/LLVM, which intends to perform the C-code static analysis. The analysis first links the code, compiles it, and performs analysis only if there are no compilation errors. It performs a whole-program analysis, i.e., it operates on the linked program. The analysis will be aborted if the code produces compilation errors and displays an error message with information about the compilation error. These compilation errors may affect the soundness of the analysis; if it does not affect the analysis's soundness, then the analysis will be continued by displaying a warning message about the compilation error. Vector Code Analyzer only supports a subset of instructions provided by LLVM, and the instructions out of this are detected, then the analysis will be aborted, and an error message will be displayed. Some of the analysis done by VCA are:

- VCA will detect if a potentially invalid pointer is used for a modifying operation
- VCA will detect the write accesses that exceed the size of the target object
- VCA will detect the usage of uninitialized variables
- VCA will detect calls to undefined functions

After the successful analysis, VCA outputs the finding to the command line. A wrapper tool then reads these findings and creates the HTML report. The findings or the final report will include the location of input files and information like the severity of the error, warning, or remark.

VCA controls the reporting of false positives through a suppressing mechanism. VCA also supports formal annotation. The user may provide additional information, such as valid ranges of parameters or global variables. The location (line in the source file) of the finding can be annotated by an instrumentation comment that suppresses any findings that occur in the same line. Suppressed findings are not entirely suppressed in the report but shown as "accepted."

4.1 VCA Framework

VCA framework is a platform on which the whole application has been developed. From the figure given below, it is clearly visible that it contains several functional entities like the front end, checks, analysis, report, and config.

Front end: It is responsible for scheduling a set of tasks, and mainly it serves as a controller for the whole framework. Some of the tasks scheduled by the front end are:

- Configuration and command-line arguments are read through the front end
- Invoke clang + llvm to parse/link the source files
- Schedules the required analysis passes
- Execute the configured internal checks
- Report findings

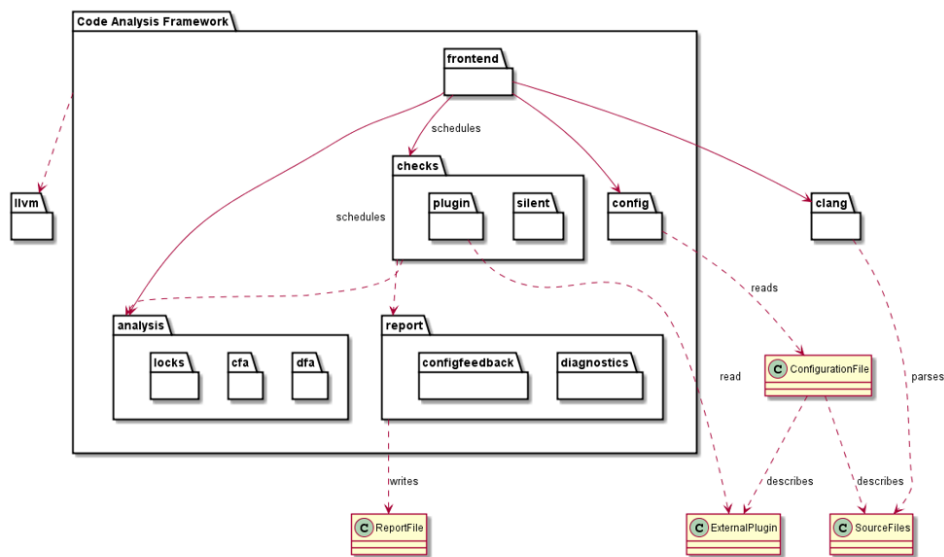


Figure 4.1: Code Analysis Framework

Config: As the word suggests, this module is meant for loading and configuring data. This module decides the source files that have to be checked, the plugins that have to be executed, and determine the need for any additional plugin-specific configuration fragments.

Analysis: This module is responsible for analyzing the control and data flow. The result of this analysis will determine the pointer and range analysis.

Checks: It is responsible for evaluating the code based on the analysis results. It will check whether the code is silent. The silence analysis checks evaluate the silence requirements by performing analysis like pointer target analysis, index analysis, undefined and uncalled Function/Variable analysis, unreachable code check. It will

prove the absence of interference concerning memory.

Report: This module collects the possible results of checks and provides it in a readable format.

4.2 Static Single Assignment Form

Static Single Assignment Form (SSA or SSA Form) is a program representation that makes the compiler optimization less complex by providing the data flow information in a unique manner [40]. The data obtained from a CFG graph with def-use is quite hard to analyze since each definition might have multiple uses and vice-versa. SSA is an intermediate representation (IR) before the actual data flow analysis. According to Andrew Appel, what exactly SSA Form represents is that each variable in the program will only have one definition, which means each variable will be assigned only once. So in order to achieve single-assignment, a new variable name have to be provided for each assignment of the variable [36].



Figure 4.2: SSA Form of a simple straight-line code

Figure 4.2 is an example of the SSA Form. From the figure itself, it is clear how an SSA form is constructed. The figure's left-hand side indicates the normal representation, and the right-hand side indicates the SSA form. For representing it in SSA, the user has to rename the variable for every new assignment and make sure there are no multiple assigning of a variable. As its name depicts Static Single assignment form, it deals only with static properties. Consider the variable a_1 , even though its value is dynamic in nature, but the static property of the variable remains unchanged during all instances (new assignment or definition of variable), that is the static property that all instances labeled a_1 refer to the same value will still hold [37].

But there arise certain situations where we have to represent not only straight-line code but also the code with branches. During these situations with branches, at some point, the branches or control flow paths will merge, and a variable may have

more than one possible point of definition, and it leads to the violation of the single-assignment property. So how can we handle this situation?

For handling such a situation, we will make use of the ϕ function. A ϕ function is of form $F \rightarrow (X, Y\dots)$, where F, X, Y are variables, and the number of operands X, Y denotes the number of control flow predecessors of the point where ϕ function occurs[51]. A ϕ function will opt for the correct definition of a variable based on the branch that was taken to enter the ϕ -node. ϕ functions are usually placed at the beginning of basic blocks, and they will be executed simultaneously before the execution of any other code in the block [37].

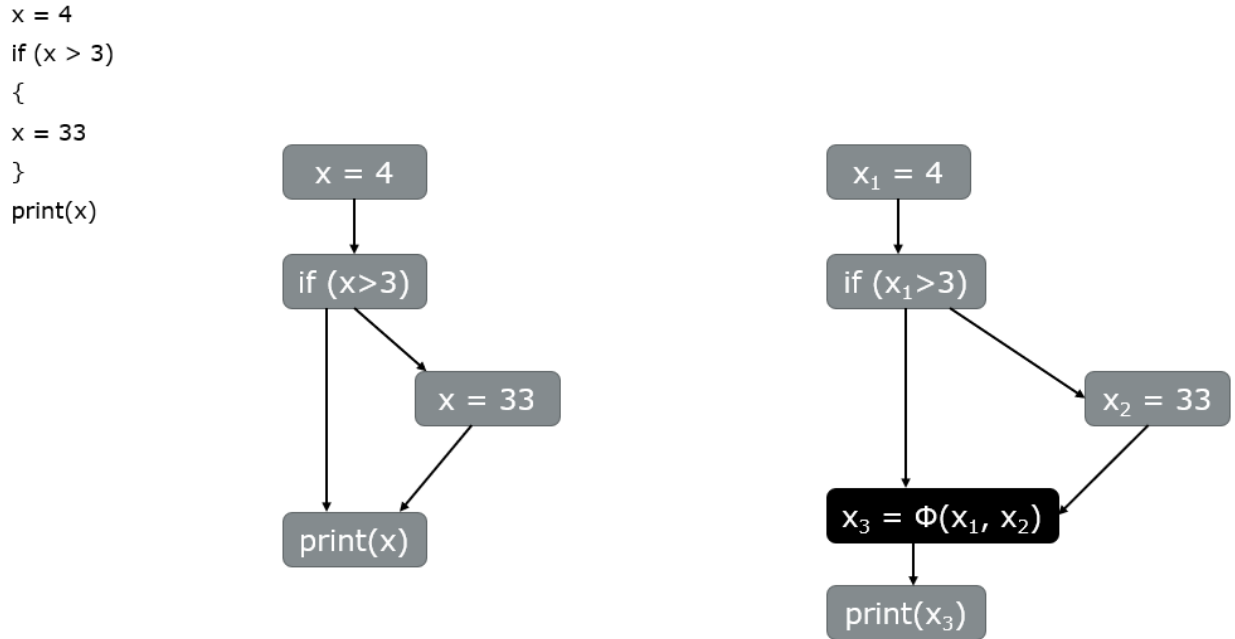


Figure 4.3: Example code (left), Control flow graph (Middle), and the control flow graph in SSA Form (Right). ϕ function and versioned variables are shown (black background)

The above figure shows a code snippet, its control flow graph, and a control flow graph in SSA Form. Here the expression $x_3 = \phi(x_1, x_2)$ is the point where branches meet, x_3 can assume either the value x_1 or x_2 it depends. x_3 will be assigned with the value of x_1 if the control obtains from the first branch, or else it will have the value x_2 . It is necessary to convert the program into SSA Form.

Some of the advantages of using the SSA Form are:

- While comparing with use-def chains, SSA chains are easy to store and update[52].
- Optimization is not accurate with the use of the use-def chain, more accurate with SSA Form[74].

- Every use of a variable is dominated by a definition of that particular variable and the efficiency of SSA optimization algorithms can be improved by making use of this property[34, 39, 67].

4.3 Abstract Interpretation

Abstract Interpretation (AI) is a framework for program analysis, which is developed by Radhia and Patrick Cousot; it will formalize the fixed-point (The fixed-point is an over-approximation of program behavior) computation using an abstract domain and abstract transfer functions[47, 48]. The formal sound approximation of a program consists of proving that its semantics (what the program executions do) satisfies its specification (what the program executions are supposed to do) [44]. It is a theory of approximation that gathers and designs approximate semantics of programs. This will be used to obtain information about programs in order to provide a sound approximation. It can be viewed as a partial execution of the program without performing all calculations and gain information about semantics. The semantics obtained can be used to specify automated program analyzers. The main aim of AI is to prove the soundness of program analysis methods where the answers will be either partial or undecidable with respect to the semantics. The abstract interpretation method is widely adopted in the design of static analyzers in order to confirm the soundness of the analysis. Abstract interpretation aims to design automatic program analysis tools for determining statically dynamic properties of programs. It is easy to design unsound and non-terminating programs, but the design of terminating and sound tools is complicated. One of the peculiar quality demands from the static analyzers is the generation of very few or no false alarms. For achieving this, abstract interpretation provides a systematic construction method based on the effective approximation of the concrete semantics, which can be (partly) automated and/or formally verified [48, 50]. The correctness of abstract interpretation depends on how efficiently the connection between concrete and abstract semantics has been made.

Some major terms concerning the abstract interpretation will be defined in the following sections.

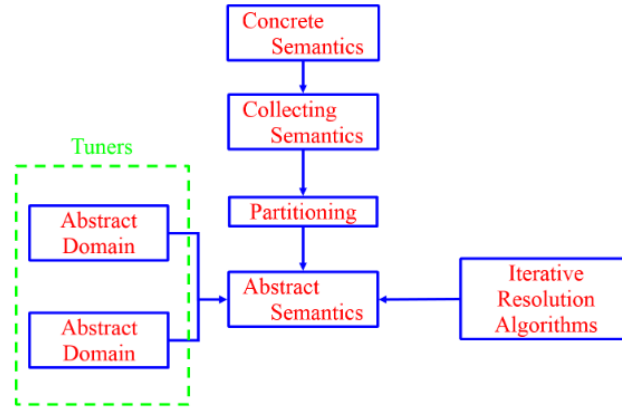


Figure 4.4: Steps of Abstract Interpretation [60]

4.3.1 Semantics

The semantics of Programming Languages unveil the most needed reasons underlying the applications of semantic techniques in computer science and introduce the mathematical theory of programming languages [55]. That means it denotes the program's mathematical meaning and describes all the possible behaviors (non-termination, termination with an error, or correct termination delivering one or more output results) of these programs when executed for all possible input data. For each part of programming language constructs, semantics is defined and connects with the mathematical representation of its meaning [41, 48]. Usually, the questions regarding the semantics are undecidable, and for that reason, it is not possible to analyze it within a finite time [58].

Concrete Semantics of a program formalizes the set of all possible executions in all possible execution environments. It is the most precise semantics, describing the program's actual execution very closely, which means it associates all the set of execution traces that will be produced during analysis.

Curves in the above Figure 4.5 represent all possible executions during the analysis; it is the abstract of standard semantics. Concrete semantics is undecidable since it is impossible to write a program to cover all the possible execution paths.

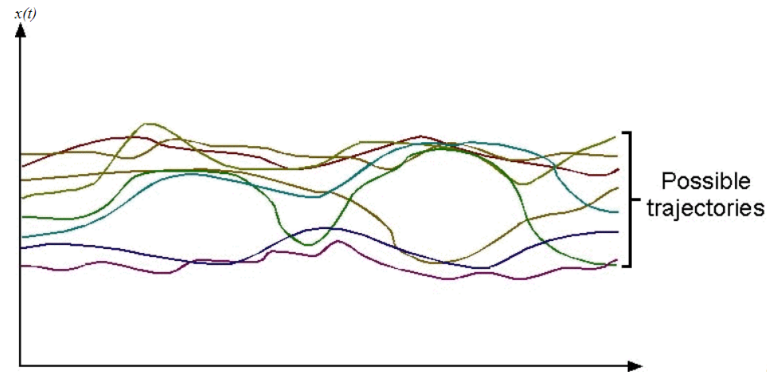


Figure 4.5: 2D Graph representing Concrete Semantics
[45]

4.3.2 Collecting Semantics

It is the first and most basic step needed for AI; all other analyses based on AI evolved from this. It computes all the memory states that occurred during the execution of the program; in the meantime, it is a tedious task to discover all the exact memory states during execution. It defines the reachable states from the initial state. It is an abstract of the standard semantics; in this, all the irrelevant data concerning the program's execution will be filtered out [35]. Examples of collecting semantics are computation traces, transitive closure of the program transition relation, set of states/predicate transformers, forward/backward reachable states [47].

4.3.3 Abstract Semantics

Abstract semantics is an approximation of concrete semantics. AI uses abstract semantics to obtain the program's essential properties; to obtain different properties, AI will use different abstract semantics. When the user wants to design an abstract semantics, we can either take the concrete semantics as the reference, use a relatively poor mathematical framework, or use the collecting semantics as the reference, and derive the abstract semantics using the Galois connection framework[35]. Consider the case of range analysis, there will be an upper and lower limit, and the properties will differ. However, as mentioned earlier, different abstract semantics will be used to obtain the upper and lower limit properties. The soundness of abstract semantics is directly related to the collecting semantics.

4.3.4 Abstract Domain

The abstract domain is the superset of abstract semantics; it contains various abstract semantics. An abstract domain is, in some way, an abstraction of the concrete semantic domain. As mentioned earlier, concrete semantics is non-computable as it is not possible to cover all the program execution path, so it is infinite and undecid-

able. To avoid this, in the abstract domain, it will not consider some of the properties of the concrete semantic domain in order to make the domain computable and computer representable. The abstract domain has to be correct because all states derived in the concrete semantic domain should also be derived in the abstract domain, but the abstract domain will also include some states that will not occur in the concrete one; users will get an over approximated result. There should then be a correspondence between the concrete semantic domain and the abstract domain to make sure that the abstract domain is actually a sound approximation of the concrete semantic domain [41].

4.3.5 Galois Connection

The Galois connection is used to establish the correspondence between the concrete and abstract domain. This corresponds to a perfect situation, where each concrete property has a unique best abstract approximation. Abstract interpretation expresses the connection between the two analyses using a Galois connection between the associated property lattices [70]. The static analysis uses Galois connections because a Galois connection determines the tightest projection of a function over one set, for example, the concrete transfer function, into another set. This projection is frequently interpretable as the optimal static analysis [68]. There exist two kinds of Galois connection; they are monotone and antitone Galois connection. However, the monotone Galois connection is mainly used for static analysis, as it keeps the precision of abstraction and concretization operation.

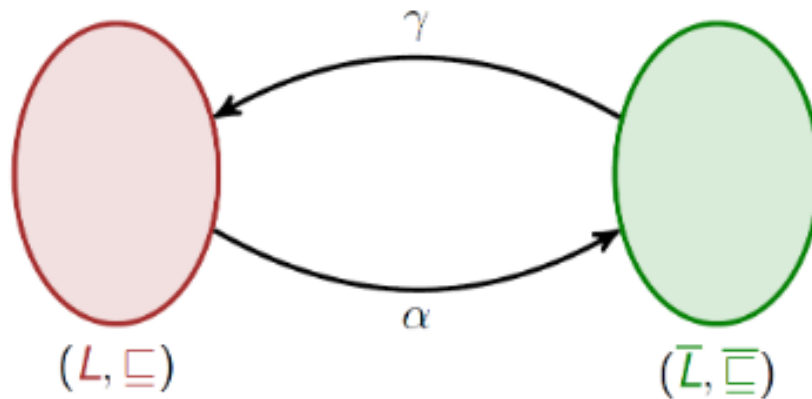


Figure 4.6: Galois Connection
[60]

The above figure shows the Galois connection between the concrete domain (red) and the abstract domain (green).

α denotes the abstraction function; it points from the concrete domain to a more abstract domain.

γ denotes the concretization function; it points from an abstract domain to a more concrete domain.

The above-shown figure 3.6 is also an example of Galois insertion, a particular case of Galois connection. It occurs when an abstract function guarantees to map all elements of abstract lattice \bar{L} .

4.3.6 Widening/Narrowing

It comes under the iterative resolution algorithm; it is one of the vital concepts of AI. The above mentioned Galois connection concept works well for a finite abstract domain but not for infinite; widening operators play a crucial role in particular when infinite abstract domains are considered to ensure the scalability of the analysis to large software systems[46]. The narrowing is used to improve the accuracy of the resulting analysis after widening.

4.4 False Alarms

False alarms may occur either because of less precision of abstract domains or due to unsoundness. There exist two types of false alarms, they are:

False Positive: Less precise or incomplete abstraction leads to false positives; it may incorrectly judge a correct statement as problematic.

False Negative: It occurs due to unsound abstractions; the program will be regarded as correct according to the specification even if it is not correct.

The above figure displays the occurrence of a false alarm, specifically a false positive. It occurred because the abstract semantics overlapped with the forbidden zone, and the concrete semantics does not lead to raising a false alarm, but in reality, no error will occur. It is a consequence of the over-approximation of program execution.

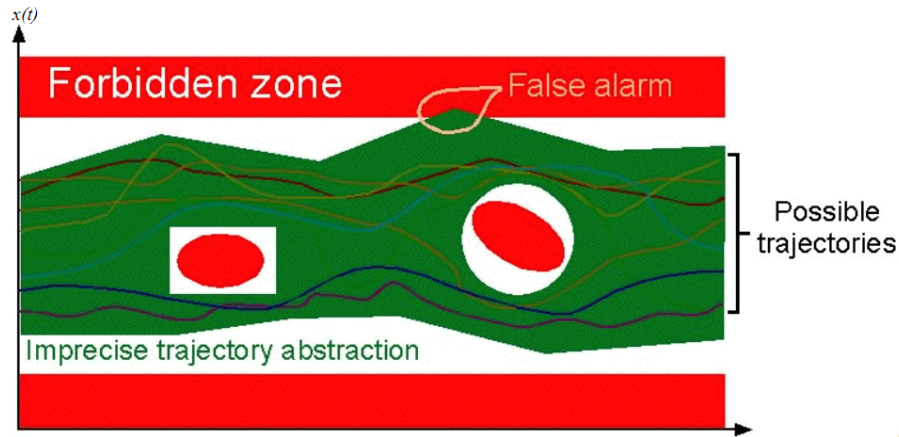


Figure 4.7: False Alarm
[45]

4.5 Sound and Unsound static analysis

A sound analyzer will analyze the code concerning a class of defects it can detect safely and exhaustively. However, sometimes soundness comes with the expense of false alarms (false positives).

An unsound analyzer might fail to raise alarms even for faulty code lines, leading to severe problems.

A sound analysis tool will never report a faulty program as correct, but it may raise a few false positives. Unsound tools will generate false-positive and false-negative alarms, affecting the program's quality. An unsound analysis tool might produce fast, mostly noise-free outputs by entirely ignoring large or difficult-to-analyze functions; this usually results in low precision. Here lies the significance of a sound analyzer; it outputs accurate and precise results, though it may contain some false positive alarms. Also, a sound tool reports the assumptions it makes so that others can verify them means which in turn helps to build trust in the tool [24].

The difference between the sound and unsound tool is shown in figure 4.8. The Green colored area depicts the area where the program is correct, and the other shows the area with a defect. The dotted circle denotes the results generated by each tool, with the programs inside the circle being reported as correct and the programs outside the circle are erroneous.

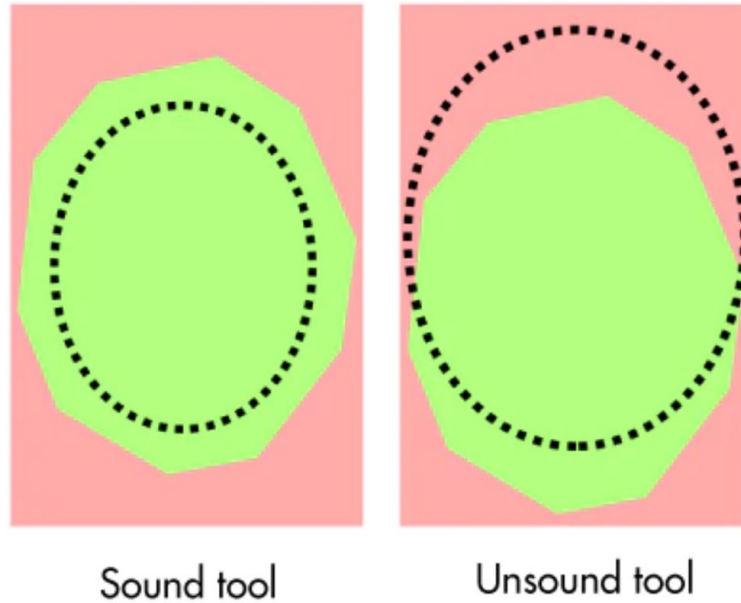


Figure 4.8: Difference between a sound and unsound tool
[32]

4.6 Data flow Analysis

A basic definition of data flow analysis is provided in Chapter 2. Data flow analysis is one of the instances of abstract interpretation, and it is done with a control flow graph, graphical representation of the program. Data flow analysis can be carried out either forward or backward or sometimes in both directions, but VCA is concerned about forward analysis only. In forward data flow analysis, the analysis begins from the entering node of CFG up to the exit node and computes input by taking the union or intersection of all of the predecessors' outputs and getting output by reasoning locally about the facts generated and killed at that point[11]. There exists some important concepts and optimization techniques while considering forward data flow analysis:

- **Reaching Definition**

Reaching definition is a data flow analysis that statically determines which definitions may reach a given point in the code. Its primary task is to identify the connection between the variable definitions and variable uses.



Figure 4.9: Reaching Definition with two nodes

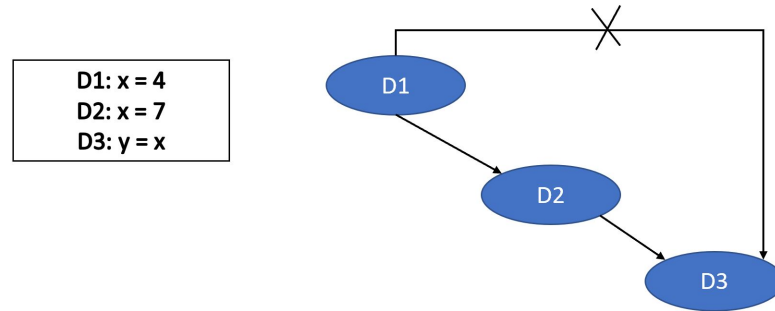


Figure 4.10: Reaching Definition with three nodes

The CFG with two nodes is an example for reaching definition as the node D1 reaches D2. However, in figure 4.10, the CFG node D1 does not reach D3 as it gets killed (definitions terminated by some other definitions in the basic block) by D2. For straight-line code, it is easier to calculate reaching definition; otherwise, need to use the iterative algorithm.

Use-Def chains (UD Chain) are computed using the reaching definition. A UD chain will contain all the use U of a variable, and its definition D , it is meant for assigning a value to a variable.

Def-use chain (DU Chain) its counterpart links each definition of the variable to those uses which that definition can reach. SSA encodes def-use information in linear space, and it is not possible to add new names to the symbol table at all assignments in SSA form. So most implementations provide def-use chains for each definition [73, 9].

• Constant Propagation

Constant propagation is the process in which the values of known constants will be substituted in expressions during compile time, which means the constants assigned to a particular variable can propagate through the CFG and can be assigned to the variable when needed.

$$\begin{aligned} a &= 21; \\ b &= a + 9; \end{aligned}$$

In the given code fragment, the value of the variable 'a' can be propagated and substituted in line 2.

```
a = 21;  
b = 30;
```

This is how the code fragment will change after constant propagation. The general approach for performing constant propagation is:

- Create the CFG for a given program.
- Relate the transfer functions(The transfer function is a mapping from one environment to the other that corresponds to the semantics of its source node) with the edges of CFG.
- Maintain the value of the variable.
- Iterate till the value of variable stabilize [66].

Sparse conditional constant propagation is a similar kind of optimization technique like constant propagation, applied in compilers after conversion to SSA form. It simultaneously removes some kinds of dead code and propagates constants throughout a program. It optimizes the code by making use of abstract interpretation of the code in SSA form [74, 43].

• Available expression Analysis

It is a forward data flow analysis problem, which determines whether an expression that has been computed previously can be reused or not. An expression ($a*b$) is said to be available at a particular point if every path from the initial node to that point must evaluate $a*b$ before reaching that particular point, and there must not be any assignments to a or b after the evaluation but before reaching that point. It's mainly used to perform Common Sub expression Elimination (CSE).

Common Sub expression Elimination is an optimization technique which is used to eliminate common or identical subexpression. For CSE, analyze the program and search for identical expressions and replace it with a single variable holding the value [67].

```
x = a + b;  
y = (a + b) * c;
```

In the given code snippet, the expression $a + b$ is repetitive, and according to CSE, it will replace the second occurrence of the subexpression with a single variable by analyzing factors like cost, storing time.

There exist two kinds of CSE, they are local and global CSE. Local CSE is applicable only for a single basic block, whereas the global CSE is meant for the whole function.

4.7 Clang/LLVM

LLVM stands for Low-Level Virtual Machine; an open-source project initially developed by Swift language creator Chris Lattner as a research project at the University of Illinois; it is an SSA-based representation capable of representing all high-level languages. It is the standard code representation used throughout all phases of the LLVM compilation strategy [20]. It provides tools for automating many of the most thankless parts of language creation: creating a compiler, porting the outputted code to multiple platforms and architectures, generating architecture-specific optimizations such as vectorization, and writing code to handle common language metaphors like exceptions [30]. The core of LLVM is the intermediate representation (IR), a low-level programming language similar to assembly. IR is a strongly typed reduced instruction set computing (RISC) instruction set that abstracts away most details of the target; it can be represented in three different forms as an in-memory compiler IR, as an on-disk bit code representation, and as a human-readable assembly language representation and all these representations are equivalent. LLVM can then compile the IR into a standalone binary or perform a JIT (just-in-time) compilation on the code to run in the context of another program, such as an interpreter or runtime for the language [30]. LLVM uses SSA form in its IR for the direct data flow, and VCA uses SSA for the indirect data flow. In VCA, it acts as a back end; on the front end, it uses Clang, which is a compiler for the programming languages like C/C++.

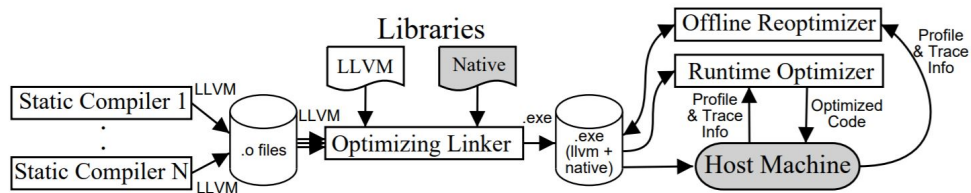


Figure 4.11: Overview of LLVM compilation strategy [59]

Clang is a compiler front end for the programming languages like C/C++. The primary design concept for clang is its library-based architecture in order to allow the compiler to be more tightly tied to tools that interact with source code, such as an integrated development environment (IDE) graphical user interface (GUI) [4]. The main advantage of using Clang over its competitors is it is faster, uses less memory, and is based on a modular design. Also, it offers more readable error and warning diagnostics, highlights the related source, and for certain kinds of common errors, it provides hints for rectifying it [19].

Figure 4.12 is an example of an error output made by Clang. From the figure itself, it


```
file.cc:6:11: error: expected ';' after  
expression  
    i += 8  
        ^  
        ;
```

Figure 4.12: output of an error

can be realized that the clang output is much more expressive than its competitors, so the user can easily understand the error and helps to fix the problem faster.

5 Existing state of Vector Code Analyzer

5.1 How Vector Code Analyzer works

The previous chapter mentioned the framework of VCA and the techniques used by VCA to perform the static analysis. This chapter will emphasize projecting how VCA analyses the code in its different stages and, finally, the output of VCA.

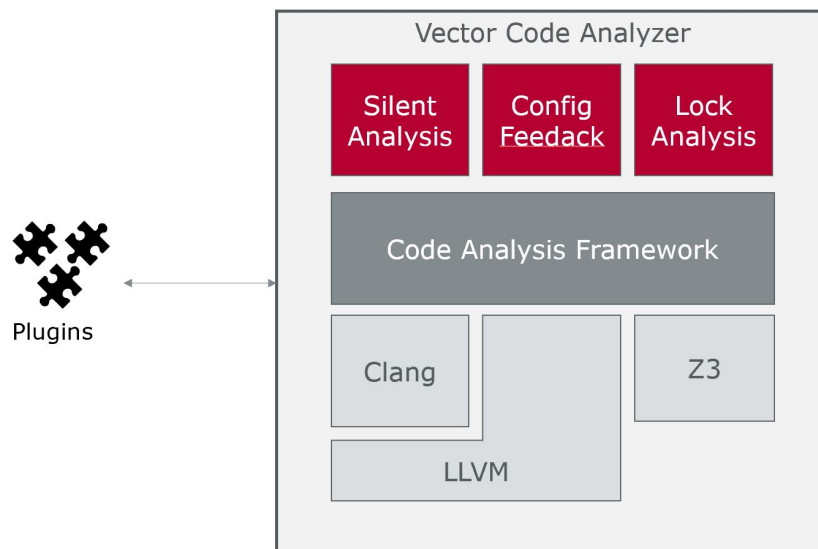


Figure 5.1: Architecture of Vector Code Analyzer

The above figure shows the architecture of VCA. VCA behaves like a compiler, and it will analyze the code only if it is error-free; also, for performing the analysis, the user needs to provide the static code files, generated source code, and finally, the paths for locating the files. Once the given code finds error-free, VCA will perform the static code analysis.

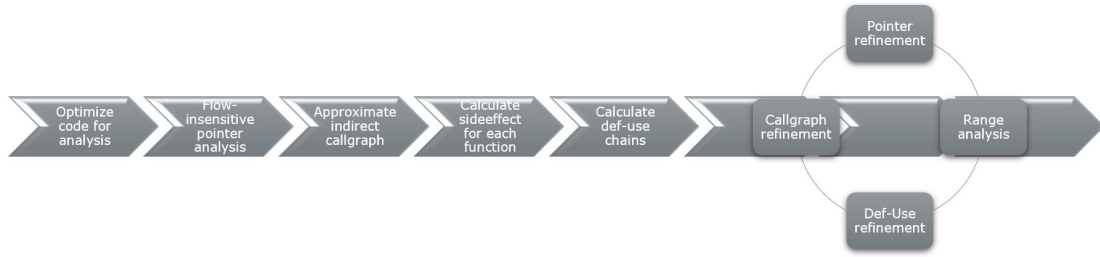


Figure 5.2: Analysis process of Vector Code Analyzer

5.1.1 Code optimization

Code optimization is the very first procedure for performing the analysis. It is done to improve the efficiency and performance of the code and the effective usage of memory space. Usually, a code is optimized by removing unwanted code lines and by rearranging the code lines. There exist several code optimization techniques like reaching definition, constant propagation, and available expression analysis. All these techniques are defined in section 4.6.

5.1.2 Flow-insensitive pointer analysis

It ignores the flow of control in a program, mostly used for whole program analysis, and computes what memory locations pointer expressions may refer to, at any time in program execution[23].

In the given code segment, it shows how a flow-insensitive pointer analysis is done. In the example, `*a` might point either to `0`, `&c`, or `&d`.

```

int b, c, d, *a;
typedef void (*fptr)(void);

void setc(void) { a = &c; }
void setd(void) { a = &d; }
void write(int v) { *a = v; } pt(a) = {0, &c, &d}

void test(void) {

    fptr f = &setc;          pt(f) = {&setc}
    f();
    write(20);
    f();

    write(30);
}

```

Figure 5.3: Flow-insensitive pointer analysis

5.1.3 Approximate indirect callgraph

The use of pointers creates serious problems for software productivity tools that use some form of semantic code analysis for software understanding, restructuring, and testing as it enables indirect memory accesses and indirect procedure calls. The call graph represents the relationship between program procedures. For example, consider an edge (P1, P2), which means procedure P1 may call procedure P2. This information is essential for program comprehension. However, such tools face a problem when the program contains indirect calls through function pointers. In this case, some form of pointer analysis may be necessary to disambiguate indirect calls[65].

In the given code snippet, the callgraph for the function test has been shown.

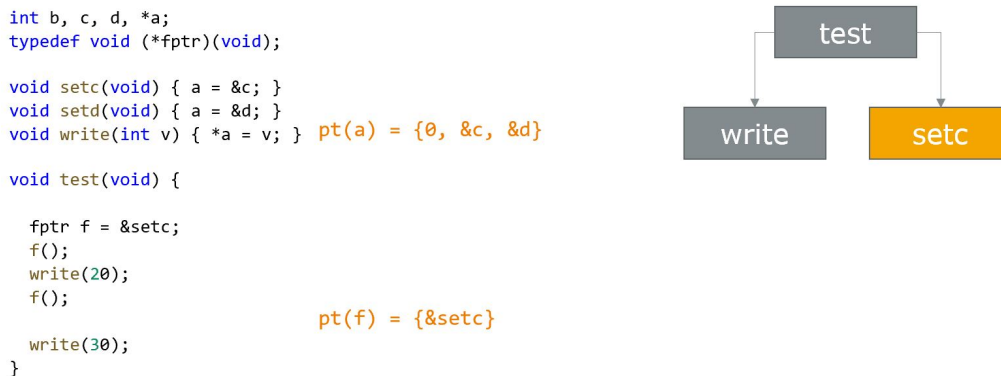


Figure 5.4: indirect callgraph

5.1.4 Calculate side effect for each function

A function or expression is said to have a side effect if it modifies a state outside its scope or has an observable interaction with its calling functions or the outside world. Some examples of side effects are:

- Modification of a global or static variable
- Modification of function arguments
- Writing data to a display or file
- Reading data
- Calling other side-effecting functions

In the presence of side effects, a program's behavior may depend on history. Understanding and debugging a function with side effects requires knowledge about the context and its possible histories[72, 57]. The given code snippet contains four

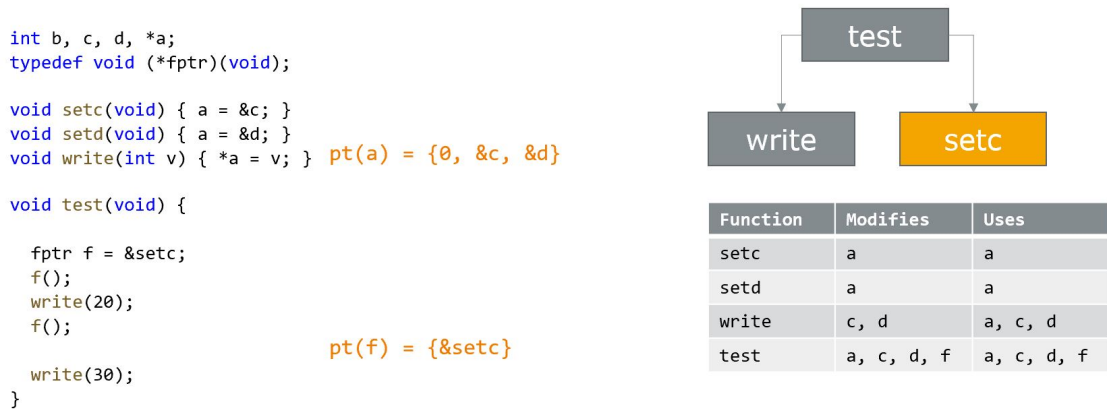


Figure 5.5: calculation of side effect

functions, and the given table shows the functions and the variables modifies or used by the functions.

5.1.5 Calculate def-use chains

The topic def-use chain is defined in section 4.6.

5.1.6 Combined data flow analysis

Combined data flow analysis is an LLVM pass that can query domain-specific values of nodes in the data flow graph. The analysis uses the generic framework to support dependencies between different analysis domains. For each domain, at least one analysis is implemented. To facilitate usage of combined DFA during abstract execution (e.g. within the condition manager), a combined DFA environment is provided that wraps the analysis manager assigned to the combined DFA. This environment just forwards requests to unbound values to the analysis manager. If the value is not yet available, an empty set is returned. Users might query the `isAllAvailable` method to check whether the environment has any unfulfilled requests.

Combined DFA comprises of:

- **Pointer refinement:** Refine approximate pointer values using refined callgraph, def-use chains, and ranges.
- **Callgraph refinement:** It calculates the caller of a function. The analysis uses the imprecise results from the inverse callgraph and restricts them by the context-sensitive results of the `pointsto` domain.
- **Def-use refinement:** It calculates the definitions of a variable. The analysis uses the imprecise results from memory SSA and restricts them by the context-sensitive results of the `pointsto` and `callgraph` domain.
- **Range analysis:** It calculates the possible ranges for variables. For any point in the type integer program, its range can be requested and is computed by the Combined DFA demand-based.

5.2 Output of Vector Code Analyzer

After the successful analysis, VCA generates its output in text format.

```

Vector Code Analyzer Version 3.4.0-0
Command Line Options: [test.c -check-silence -print-module -messa-ignore-undef -print-pass=messa-ignore-undef -memory-phi-names -combined-dfa-c
Target: i686-pc-windows-gnu

Compiling...
Linking...
Module built without errors.
Used source files:
  test.c

Analyzing...
C:\Users\visamj\Desktop\vca\test.c:29:7: error: invalid write access to unknown pointer
    a[i] = 123;
      ^
C:\Users\visamj\Desktop\vca\test.c:18:2: note: when called from here
  write(&a[0], i); // <-- range of i 4..20 when called from test.c:21 and 22..20 when called from test:22 last modified in test.c:7:
  ^
C:\Users\visamj\Desktop\vca\test.c:41:2: note: when called from here
  context1(4);
  ^
C:\Users\visamj\Desktop\vca\test.c:29:7: error: invalid write access to unknown pointer
    a[i] = 123;
      ^
C:\Users\visamj\Desktop\vca\test.c:18:2: note: when called from here
  write(&a[0], i); // <-- range of i 4..20 when called from test.c:21 and 22..20 when called from test:22 last modified in test.c:7:
  ^
C:\Users\visamj\Desktop\vca\test.c:42:2: note: when called from here
  context1(22);
  ^
C:\Users\visamj\Desktop\vca\test.c:29:2: error: potential pointer offset (20) exceeds bounds ([0,19])
    a[i] = 123;
      ^
C:\Users\visamj\Desktop\vca\test.c:18:2: note: when called from here
  write(&a[0], i); // <-- range of i 4..20 when called from test.c:21 and 22..20 when called from test:22 last modified in test.c:7:
  ^
C:\Users\visamj\Desktop\vca\test.c:41:2: note: when called from here
  context1(4);
  ^
C:\Users\visamj\Desktop\vca\test.c:29:2: error: potential pointer offset (22) exceeds bounds ([0,19])
    a[i] = 123;
      ^
C:\Users\visamj\Desktop\vca\test.c:18:2: note: when called from here
  write(&a[0], i); // <-- range of i 4..20 when called from test.c:21 and 22..20 when called from test:22 last modified in test.c:7:
  ^
C:\Users\visamj\Desktop\vca\test.c:42:2: note: when called from here
  context1(22);
  ^

```

Figure 5.6: Output of VCA

This is how the output of VCA looks like. The output window consists of all the necessary information for the user after the successful analysis. It provides information regarding the issues like array out of bounds, unknown pointer, and more. However, the main problem lies in understanding the root cause of these issues. Since the output window is not responsive, and even though it shows the issue, it is a tiresome job for the user to find the exact locations of the issue in code and the root cause. Because the output window doesn't provide any additional information other than

the type and location of the issue, also it's impossible to know the severity of these issues and not possible to comment about these issues in VCA.

So due to the existing scenario, the user is forced to do a manual analysis to understand the issues in the code even after the successful analysis using VCA. This causes much burden to the user as the complexity of analysis increases exponentially with the number of code, and there is even a possibility for skipping some of the mistakes while analyzing manually. Due to these limitations, VCA cannot fulfill its intended purpose. So how these issues directly affecting the users?

5.3 Issues concerned with the users

The primary benefits users expect from the automated static analysis are efficiency, accuracy, and less time consumption. Even after the successful analysis, there arise certain situations where the user has to perform manual analysis to verify whether the tool's findings are correct.

Some of the major hurdles faced by users during manual and automated analysis are:

- Time-consuming: Manual analysis took a lot of time and needed expertise for the analysis; otherwise, it would not be effective.
In the case of analysis using automated tools, it is possible to perform the analysis within a short span, but in the end, the user has to analyze the result.
- False positives/negatives: Manual analysis by an expert might probably give less false positive/negative issues.
Whereas in the case of automated tools as well, users have to verify each issue, and if a tool generates more false positives, it might lead the user to skip specific issues and affect the accuracy of the result. No tool will produce zero false positive/negative, but they tried to make this issue minimal.
- Code coverage: With the automated analysis, it is possible to analyze the source code broader, faster, and it is possible to repeatedly perform the analysis without much effort. The analysis with a tool can be performed by a user with minimal knowledge in the field of development.
In manual analysis, it is really hard to make sure whether full code coverage has been gained, and as mentioned earlier, it is a time-consuming process.

These are some of the general concerns faced by the users. It is clear that for performing the static code analysis, it is always better to have an efficient tool and a manual analysis of the result. By this combination, it is possible to ensure the efficiency and quality of the code. It is not wise to entirely depend on automated tools or manual analysis because both have their limitations.

Manual analysis requires a large amount of the developer's time, and the project can be delayed for days or weeks. The chance of negligence or skipping of the issue

is also there, which might cause some serious issue once the software is delivered. It will cost huge to rectify those issues in the later stage, and the impact of these issues will also vary on the type of software/code that has been analyzed.

Like all software, static analysis tools are a collection of trade-offs. If they go for speed, their analysis's depth suffers and get more false positives. Once they try to reduce the false positives, they run slower, and in the case of inexpensive tools, they might have less expertise and less original research behind them. One tool may be very good at catching some classes of bugs, and another tool may be good at catching other classes of bugs; none are likely to be good at catching all classes of bugs. These trade-offs will affect the tool results[62]. From these, it is clear even if the user uses the most efficient tool available in the market it has to be follow up with manual analysis. The reviewer definitely have the knowledge about the tool and aware of the pros and cons of the tool. This will help the reviewer to weed out the problems before they ever waste a developer's time by shielding them from the noise that all static analysis tools create.

So how can we improvise the efficiency and accuracy of Static code analysis? From the data that has been analyzed, the result made by the most efficient static analysis tool has to be analyzed manually. It is possible to reduce the complexity of manual analysis to a certain extent if the code becomes more transparent, and it can be achieved by the visualization of data flow. VCA is also an efficient static analysis tool, and the major drawback faced by VCA is in the complexity and effort required for analyzing the output generated by it.

6 Concept

So how is it possible to reduce the complexity of manual analysis of output, ensuring the code's maximum efficiency and quality? The reviewer's vast amount of time is in understanding the root cause of the issue and data flow. Though the VCA provides the location and cause of the problem, the reviewer has to traverse across the code to determine the actual cause of the issue, and the reviewer has to understand the exact data flow, which is practically impossible. However, there is a solution to overcome this issue by making the code more transparent by visualizing the data flow, which will allow the reviewer to understand the code better and discover the root cause of problems.

6.1 Output window

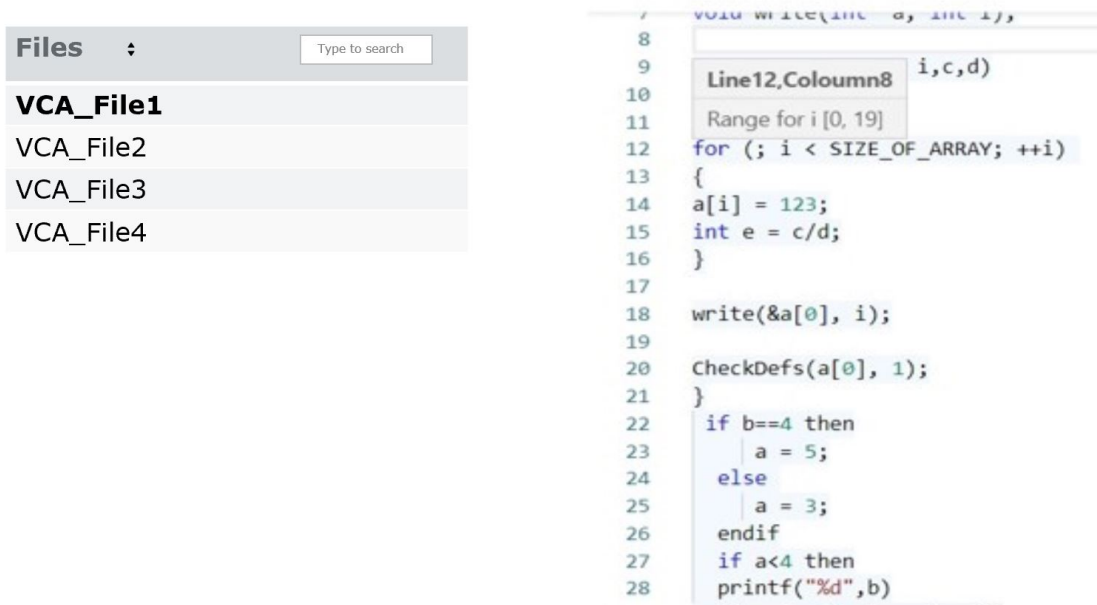


Figure 6.1: Conceptual output window

The main limitation caused by VCA is its unresponsive output window. The conceptual output window shown in figure 6.1 will help overcome the limitations of existing output by VCA to a specific limit.

The left side of the output window contains the data table displaying the file name.

It comes with a search bar as well as a sort option. While selecting a particular file, the data on the file will get displayed on the editor pane, and the chosen file gets highlighted on the table, so the user will be able to identify the selected file. Once the file is selected, it will be displayed on the editor pane, and the reviewer can go through the file. The thesis considers the evaluation of single-page code only. The search bar helps to find the file easily if there are more files by just typing the required file's name and the sorting button helps to sort the files. The code displayed on the editor pane is responsive, while hovering over the data variable, it will display the range of the variable, and while clicking on a particular data variable, the data flow of that selected variable will get displayed on another window.

6.1.1 Range analysis of variable

The range of a data variable is also shown in the existing result window of VCA. However, the result window is in text format. Contrary to the existing result window, here while hovering over the variable, a pop up will get displayed, and it shows the range of the variable, and its location. So how the range of variables is determined. For determining the range of the variable, VCA uses an interval set. The precision is gained over representing possible integer values by a single interval. Such a single interval can be represented as a `llvm::ConstantRange`, which consists of a lower and an upper bound. Regarding stability, an unstable interval set means that the interval set size increases after evaluating compared to the last result of the analysis. Once a fixed point is reached and the intervals of a node are determined, the interval set is stable and is the only relevant result. Since the `CombinedDFA` computes an over-approximation, an interval set computed for an integer variable of a program has the following meaning: during the program's execution, the variable can only assume values contained in the intervals. States in which a variable would assume a value not contained in its intervals are unreachable. Due to the over-approximation, it is, however, possible that there are elements contained in the intervals that can never actually be assumed in any run of the program. It is necessary to make an accurate approximation and need to avoid this kind of over approximation or under approximation. This can be achieved by abstract interpretation.

The precise range of the variable is essential in compiler optimization, program checking, and computing the minimum bit-width requirement. The choice of abstract domain and the mapping between its members and the concrete values creates a trade-off between the quality of results and the analysis running time[47]. Abstract interpretation is not perfect, but for fixed point computations, it is a perfect choice. William Harrison does the first work related to the variable value range using abstract interpretation by proposing a framework that combined two mechanisms, and they are range propagation and range analysis. This helps to avoid the potentially costly fixed-point computation on looping constructs[56].

- Range propagation: It is a simple algorithm that uses the data and the conditional structure of a program to derive and propagate refinements in the

accuracy of range information. First, it transforms the analyzed program into the Static Single Assignment (SSA) form. A range is defined for each point in the program, and the algorithm iteratively computes refinements of range information using specially defined transfer functions that operate on ranges. This algorithm is effective on straight-line code, unfortunately this algorithm is not effective in the case of loops[56].

- Range analysis: It is an algorithm that tracks the changes to a variable at each point of the program but does so by avoiding the loop's conditional structure. The information obtained is then used as a base for induction to derive a range of values for the variable. Ignoring conditional branches in the loop structure limits the accuracy of this technique for range analysis. However, if the problem is to identify unused or compile-time bound bits in a variable, this result with limited accuracy still may suffice[56].

The results obtained from range propagation and range analysis are combined, and the required approximation of range is made.

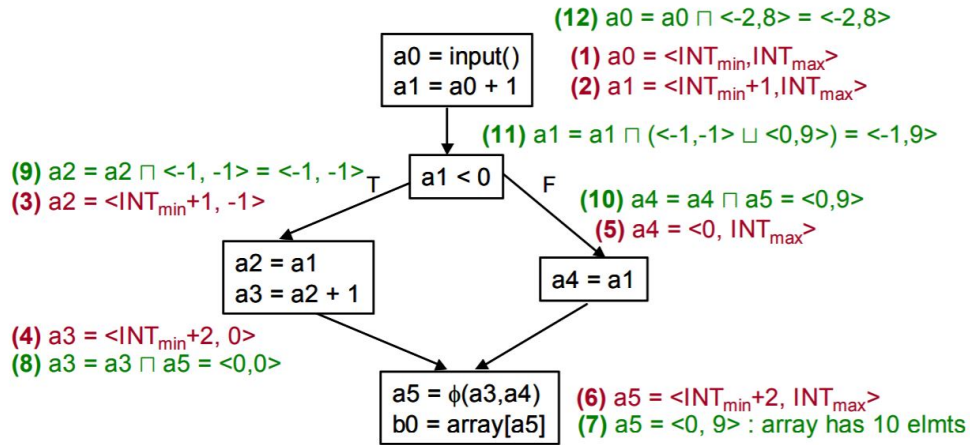


Figure 6.2: Example for range propagation
[64]

Steps 1 - 6 in figure 6.2 denotes the forward, and 7 - 12 denotes the backward range propagation. Forward range propagation is done using Breadth-First search, and it will carry out until a fixed point, or exit point is reached. Backward range propagation is done to refine the range values obtained from forward range propagation.

6.2 Visualization of straight-line code

As mentioned earlier, the output window is responsive, and while clicking on a particular variable, it will display the data flow for that particular variable.

```

void main()
{
    int x = 10, y = 15, temp;
    temp = x;
    x = y;
    y = temp;
    printf("x = %d and y = %d", x, y);
}

```

Figure 6.3: Simple C code for swapping

The above code segment shows a simple C code for swapping. It is a straight-line code means a section of a program in which instructions are executed sequentially without jumps, branching, or looping. The code segment variable 'x' is highlighted; how can the reviewer determine the data flow of variable x. They usually have to perform the manual analysis, and for this particular code segment, it can be done quickly. However, it will not be the same for all scenarios depending on the number of code lines and usage of the variable.

The complexity can be reduced by visualizing the data flow of the variable. So in the above-given code fragment reviewer have to identify the data flow of variable x, for that click on the variable x in the output window, while clicking on the variable will redirects the reviewer to an alternate output window, there it will show the significant code lines corresponding to the variable and the output window comes along with a slide bar and two buttons(Prev, Next). The user can click on or drag the mouse over the slider bar to jump to a particular point or use the VCR-style navigation buttons to step forwards and backward over the code lines. This will allow the user/reviewer to understand the code better and the data flow of a particular variable. Instead of providing the entire data flow of the code, the user can access the data flow of the concerned variable, which will also help save time. Because if there is an error, the output window will provide the exact location and cause of the

6 Concept

error, then in order to determine the root cause or to do an analysis, what the user has to do is to click on the concerned variable and analyze the data flow, this will provide a better insight to the code and reduce the time and complexity of manual analysis. Figures 6.4 and 6.5 show how the data flow analysis of straight-line code

```
➔ int x = 10, y = 15, temp;

    temp = x;

    x = y;

    printf("x = %d and y = %d", x, y);
```




Figure 6.4: Initial step of data flow analysis

```
int x = 10, y = 15, temp;

    temp = x;

    x = y;

➔ x = 15 printf("x = %d and y = %d", x, y);
```

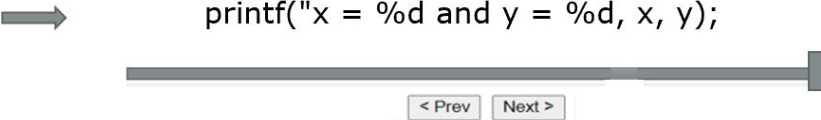


Figure 6.5: Final step of data flow analysis

has been made. Here as shown in both figures, while clicking on the variable, it takes the user to the alternate output window; the corresponding code lines related to the variable will get displayed; here in the figure, it is variable x. Only the relevant section of the code will get displayed here, and the user can visualize the data flow with the help of navigation buttons and slide bar and understand the data flow with ease. The arrow mark will also provide the user a better overview of the data flow,

and as shown in the figure, with the help of codelens user can see the value of the variable while traversing across the lines as shown in figure 6.5. The relevant data regarding the data flow will get from VCA after the successful analysis.

Code folding is a technique, which allows the user to fold and unfold certain parts of code selectively. This allows the user to manage large amounts of text while viewing only those subsections of the text that are specifically relevant at any given time. For example Functions, there might be certain situations when there is no need to display the whole function; in such cases, it is better to use the folding technique, which will provide a better view of the resulting output window.

```

1 > void context1(int i,c,d)...

1 void context1(int i,c,d)
2 {
3   b = 0;
4   if b==4 then
5     a = 5;
6   else
7     a = 3;
8   endif
9   if a<4 then
10  }
```

Figure 6.6: Example for folding

The topmost portion of the figure displays the folded function, and the other portion shows the unfolded function section.

6.3 Visualization of loops

Above mentioned techniques work well with straight-line code, but it is not sufficient for conditions like loops, branches, and decidability problems. For straight lines code, the user can seamlessly traverse across the code lines and understand the data flow, but for codes having loops or some conditions, it will be hard to understand and consumes much time for analysis.

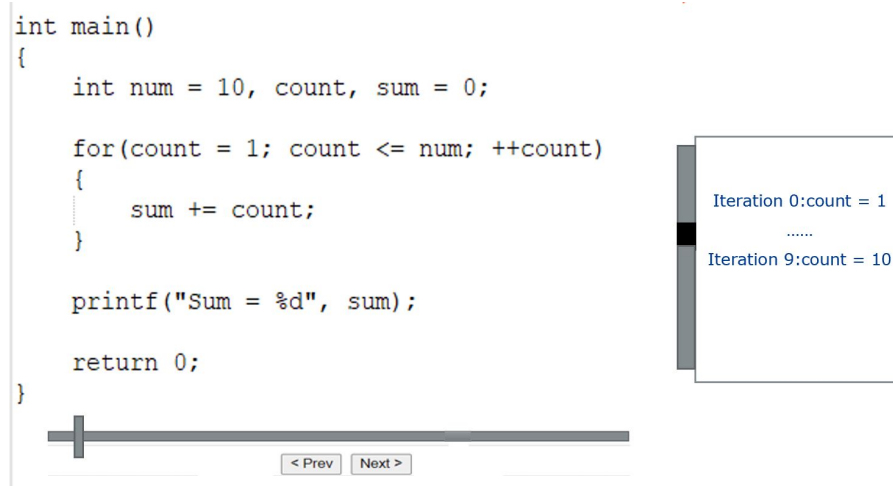


Figure 6.7: concept for loops

In figure 6.7, a code segment with a simple for loop is shown. After the combinedDFA and range analysis, an approximate range will be obtained, and this data has been fed to the result window. But how the user can make use of this data in a useful manner. If the previously used method is used here, then the user has to traverse across the code for n times depending on the loop limit while analyzing the result. That means the user does not get many benefits. In order to overcome this, apart from the slide bar and buttons introduced a popup box. So whenever the user encounters a loop, a popup will be displayed, and the user will get to know the number of iterations and either skip all these iterations or else they can choose the desired iteration step or one which is causing the issue.

For displaying the iterations narrowing operator of abstract interpretation has a significant. The interval set received after the range analysis has to be trimmed, as displayed in figure 6.7. With the widening/narrowing operator's help, it is proved that infinite abstract domains can lead to useful static analyses for a given programming language that is strictly more precise and equally efficient than any other one using a finite abstract domain or an abstract domain satisfying chain conditions[48]. As previously mentioned in section 4.3.6 narrowing operator is used to increase the accuracy of the result generated after widening because, in many cases, it overshoots and provides improper results. Consider the following code:

```
x = 1;
while(*) {
x = 2; }
```

In the given code, after widening x 's abstract value will become $[1, \infty]$, but the more accurate value is $[1, 2]$. That is narrowing operator will get iterate until it reaches stabilization or after reaching the maximum number of iterations. After every iteration solution get more precise, but it works only for monotonic (for a bigger set of states, the constant that includes all the states does not become smaller) functions[53, 69].

6 Concept

Almost a similar kind of representation is used to analyze decidability problems and code with branches. It will help the users if the complex/confusing part of the code can be represented as flow charts.

```
1 int a = 0;
2 void Decide(int x)
3 {
4   if (x>0)
5   {
6     a = 3 ;
7   }
8   else
9   {
10    a = 7;
11  }
12 }
13 int getA()
14 {
15  return a;
16 }
17 int main (unsigned v)
18 {
19  if (v < 15)
20  Decide( v);
21  int x = getA();
22 }
```

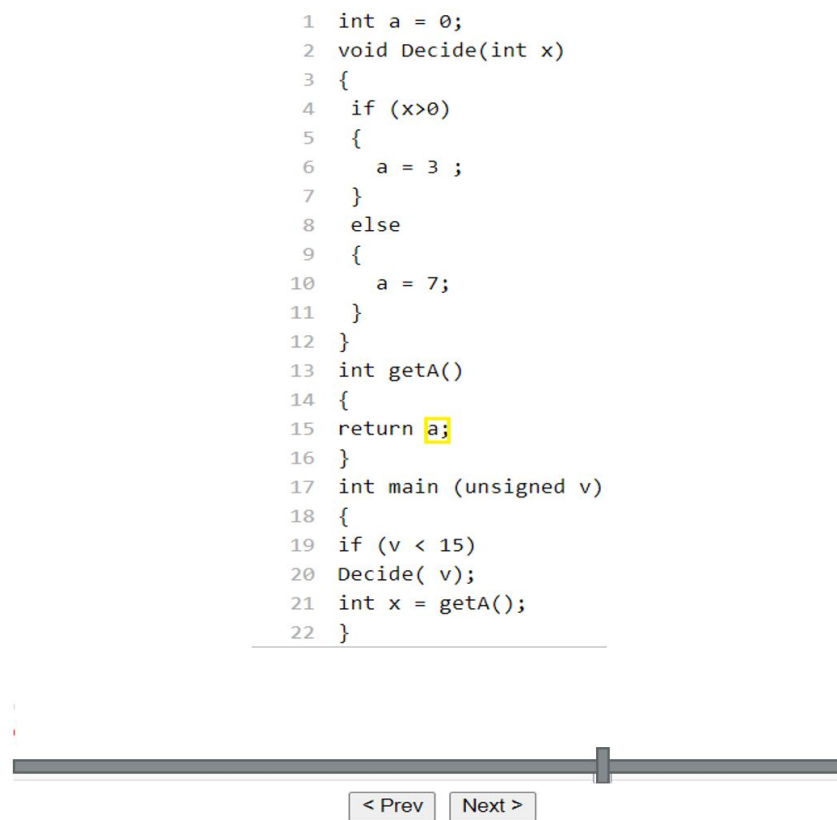


Figure 6.8: Code with decidability problem

The code segment given in the code highlights one of the most critical concerns during the manual analysis of the output. Consider the variable 'a' that is highlighted in the code segment. What might be the value of that variable? Its value depends on the condition at line number 4. Depending on that condition, the value of a variable can be either 3 or 7. There is only one condition in the given example, and it can be understood from the code without much difficulty. However, the difficulty increases exponentially with the number of conditions.

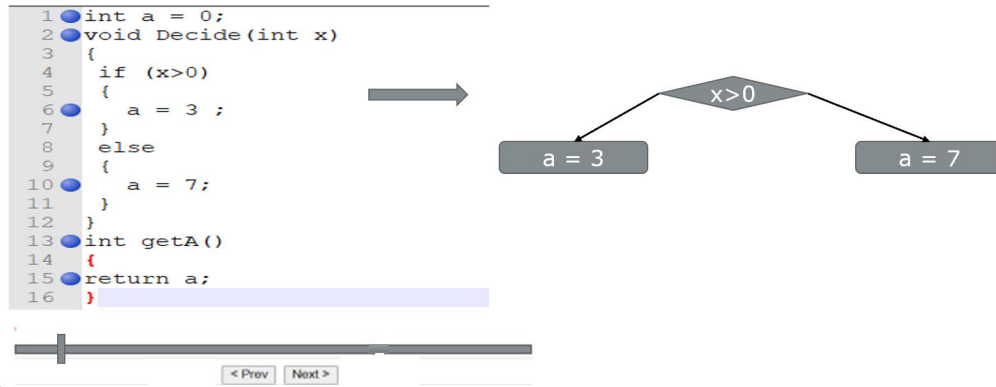


Figure 6.9: Concept for the decidability problem

A flowchart representation will give the user better clarity in this kind of situation. So here, while clicking on the variable 'a', it will redirect the user to the alternate output window having the code lines that are associated with that specific variable, and when enters at the condition during the analysis, the flow chart representation of the condition will get displayed and helps the user to understand the possible value that the variable will hold.

6.4 Environment preparation

6.4.1 Vue.js

The whole concept is developed on the Vue.js framework. It is an open-source modelviewview model front-end JavaScript framework for building user interfaces and single-page applications. Unlike other monolithic frameworks, Vue is designed from the ground up to be incrementally adoptable. The core library is focused on the view layer only and is easy to pick up and integrate with other libraries or existing projects[63, 31]. Prerequisites for installing Vue.js are:

- Nodejs: Nodejs is a JavaScript run time built on Chrome's V8 JavaScript engine for front-end development. Nodejs uses npm as the package manager, which helps maintain the external JS library dependencies used in the GUI development[21].
- Vue CLI: It is the standard tooling baseline for the Vue ecosystem. It ensures the various build tools work smoothly together with sensible defaults[31].
- Visual Studio Code: It is a free source code editor; it supports Vuejs.
- Browser: It is necessary to view the GUI developed using Vuejs and debugging if there are any issues related to development.

6.4.2 Monaco editor

Monaco editor is the editor used during the project creation. It is the code editor that powers the VS code. It is easy to set up the Vuejs project, and it has a handful of features. For installation, please refer to the official website[28].

7 Conclusion

This thesis provides concepts for visualizing the data-flow to reduce the complexity of analyzing the output window VCA, a static code analysis tool. These tools' main goal is to ensure that the coding and compliance standards are met and to point out errors. The ineffective static analysis may create several issues as it is being carried out in the early development stage. The issues that remained unnoticed or skipped will affect future code development, and it becomes tough to figure out the issue; it is almost impossible to recreate the problem. The rectification of these errors will cost more and consumes much time.

With data-flow visualization, users will better understand the code, and the analysis of output becomes less complicated as the code gets more transparent. The effective static code analysis by VCA and the concepts mentioned in this thesis will result in the analysis's betterment. The major hurdle lies in analyzing the tool's results, and by visualizing the data-flow and making the result window responsive, the user can analyze the result with much ease. After the successful analysis, the user can analyze the result without any problem as the mentioned concepts will help the user identify the root cause, find the data-flow of a variable, and the overall understanding of the code. This will help perform result analysis with time efficiency and reduce the risk of finding the error's root cause.

8 Future Scope of Work

This thesis work will pave the way for several exciting research opportunities. The topic visualization of data flow itself is a unique research topic with many possibilities regarding the static analysis. Several tools and methods are used to analyze the code, but none of the methods are efficient enough to understand the data-flow, which is essential to understand the code. This thesis is an elementary work that aims to provide a concept for the visualization of data flow. It focused mainly on visualizing the data flow of content from a single file, so it is possible to extend the work to visualize data flow having multiple files. This is one of the most needed and interesting extension in the case of visualization of data flow. In most cases, static code analyzers have to analyze solutions having multiple files, and a concept for this will help the developers in the future and reduce the complexity in understanding the code.

The loops and conditions are visualized as flowcharts; for a single file code, it is manageable, but when dealing with multiple files, it is better to find an alternate mechanism for displaying the visualization instead of a popup.

References

- [1] Analysis types, <https://docs.parasoft.com/display/JTEST1042/Analysis+Types+1>
- [2] The astrée analyzer, <https://www.astree.ens.fr/>
- [3] C/c++ static code analysis tool, <https://www.parasoft.com/ctest/static-analysis/>
- [4] Clang - features and goals, <https://clang.llvm.org/features.html#libraryarch>
- [5] Code analysis: Enhance code reviews and traditional testing, <https://www.castsoftware.com/glossary/code-analysis#:~:text=Code%20analysis%20is%20the%20analysis,%2Dto%2Dbe%20deployed%20software.>
- [6] Configuring test configurations, <https://docs.parasoft.com/display/DTP541/Configuring+Test+Configurations>
- [7] Control-flow graph, https://en.wikipedia.org/wiki/Control-flow_graph
- [8] The current state of automotive software related recalls, <https://medium.com/@SibrosTech/the-current-state-of-automotive-software-related-recalls-ef5ca95a88e2>
- [9] Data flow analysis and optimization, <https://www.cs.princeton.edu/courses/archive/spring03/cs320/notes/ssa.pdf>
- [10] Data flow analysis in compiler, <https://www.geeksforgeeks.org/data-flow-analysis-compiler/>
- [11] Dataflow analysis, <https://courses.cs.washington.edu/courses/cse401/16wi/sections/section8/dfa.html>
- [12] Division by zero, <https://www.viva64.com/en/t/0085/>
- [13] Dynamic code analysis, <https://www.viva64.com/en/t/0070/>
- [14] Dynamic code analysis, https://en.wikipedia.org/wiki/Dynamic_program_analysis
- [15] The explosion of the ariane 5, <https://www-users.math.umn.edu/~arnold/disasters/ariane.html>

REFERENCES

- [16] Fast and sound runtime error analysis, <https://www.absint.com/astree/index.htm/>
- [17] Flow analysis, https://docs.parasoft.com/display/_CPPDESKE1033/Flow+Analysis#FlowAnalysis-UnderstandingFlowPaths
- [18] How does static analysis prevent defects and accelerate delivery?, <https://dzone.com/articles/how-does-static-analysis-prevent-defects-and-accel>
- [19] An introduction to clang, <https://www.ics.com/blog/introduction-clang>
- [20] Llmv language reference manual, <https://llvm.org/docs/LangRef.html>
- [21] Node.js, <https://nodejs.org/en/>
- [22] Null pointer, http://oer2go.org:81/wikipedia_en_all_novid_2017-08/A/Null_pointer.html
- [23] Pointer analysis, <https://www.seas.harvard.edu/courses/cs252/2011sp/slides/Lec06-PointerAnalysis.pdf>
- [24] Sound static analysis for security workshop, <https://www.adacore.com/sound-static-analysis-workshop/program/chapman-talk>
- [25] Static code analysis, https://owasp.org/www-community/controls/Static_Code_Analysis
- [26] Static code analysis for c and c++, <https://www.perforce.com/products/helix-qac>
- [27] Static runtime error analysis with astrée, <https://www.absint.com/astree/workflow.htm>
- [28] Visual studio code on windows, <https://code.visualstudio.com/docs/setup/windows>
- [29] What is iso 26262?, <https://www.synopsys.com/automotive/what-is-iso-26262.html>
- [30] What is llvm? the power behind swift, rust, clang, and more, <https://www.infoworld.com/article/3247799/what-is-llvm-the-power-behind-swift-rust-clang-and-more.html>
- [31] What is vue.js?, <https://vuejs.org/v2/guide/#What-is-Vue-js>
- [32] What's the difference between sound and unsound static analysis?, <https://www.electronicdesign.com/technologies/embedded-revolution/article/21806987/whats-the-difference-between-sound-and-unsound-static-analysis>

REFERENCES

- [33] When smart ships divide by zero stranding the uss yorktown, <https://medium.com/dataseries/when-smart-ships-divide-by-zero-uss-yorktown-4e53837f75b2>
- [34] Alpern, B., Wegman, M.N., Zadeck, F.K.: Detecting equality of variables in programs. In: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 1–11 (1988)
- [35] Amato, G., Meo, M.C., Scozzari, F.: On collecting semantics for program analysis. *Theoretical Computer Science* (2020)
- [36] Appel, A.W.: Ssa is functional programming. *ACM SIGPLAN Notices* 33(4), 17–20 (1998)
- [37] Aycock, J., Horspool, N.: Simple generation of static single-assignment form. In: International Conference on Compiler Construction. pp. 110–125. Springer (2000)
- [38] Bellairs, R.: What is static analysis (static code analysis). Perforce Software (2020)
- [39] Bowen, J., Appel, A.W.: Modern compiler implementation in c: Basic techniques, modern compiler implementation-in java: Basic techniques, modern compiler implementation in ml: Basic techniques. *Computers and Chemistry* 22(2), 265 (1998)
- [40] Buchwald, S., Lohner, D., Ullrich, S.: Verified construction of static single assignment form. In: Proceedings of the 25th International Conference on Compiler Construction. pp. 67–76 (2016)
- [41] Bygde, S.: Abstract Interpretation and Abstract Domains with special attention to the congruence domain. Ph.D. thesis, Masters thesis, Mälardalen University, Sweden (2006)
- [42] Chelf, B., Ebert, C.: Ensuring the integrity of embedded software with static code analysis. *IEEE software* 26(3), 96–99 (2009)
- [43] Click, C., Cooper, K.D.: Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 17(2), 181–196 (1995)
- [44] Cousot, P.: Abstract interpretation, <https://www.di.ens.fr/~cousot/AI/#sec:libraries>
- [45] Cousot, P.: Abstract interpretation in a nutshell, <https://www.di.ens.fr/~cousot/AI/IntroAbsInt.html>

REFERENCES

- [46] Cousot, P.: Proving the absence of run-time errors in safety-critical avionics code. In: Proceedings of the 7th ACM & IEEE international conference on Embedded software. pp. 7–9 (2007)
- [47] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 238–252 (1977)
- [48] Cousot, P., Cousot, R.: Abstract interpretation frameworks. *Journal of logic and computation* 2(4), 511–547 (1992)
- [49] Cousot, P., Cousot, R., Feret, J., Antoine, M., Mauborgne, L., Monniaux, D., Rival, X.: Varieties of static analyzers: A comparison with astrée. In: First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE'07). pp. 3–20. IEEE (2007)
- [50] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The astrée analyzer. In: European Symposium on Programming. pp. 21–30. Springer (2005)
- [51] Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: An efficient method of computing static single assignment form. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 25–35 (1989)
- [52] Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13(4), 451–490 (1991)
- [53] Dillig, I.: Abstract interpretation, <https://www.cs.utexas.edu/~isil/cs389L/AI-6up.pdf>
- [54] Enrico Perla, M.O., et al.: A guide to kernel exploitation. Syngress, (2011)
- [55] Gunter, C.A.: *Semantics of programming languages: structures and techniques*. MIT press (1992)
- [56] Harrison, W.H.: Compiler analysis of the value ranges for variables. *IEEE Transactions on software engineering* (3), 243–250 (1977)
- [57] Hughes, J.: Why functional programming matters. *The computer journal* 32(2), 98–107 (1989)
- [58] J. Bertrane, F. Camporesi, P.C.R.C.J.F.V.L.A.M.X.R.A.R.M.Z.: Abstract interpretation and semantics, <https://www.di.ens.fr/~cousot/Equipeabsint-eg.shtml>

REFERENCES

- [59] Lattner, C., Adve, V.: The llvm instruction set and compilation strategy. CS Dept., Univ. of Illinois at Urbana-Champaign, Tech. Report UIUCDCS (2002)
- [60] Le, W.: Abstract interpretation, <http://web.cs.iastate.edu/~weile/cs513x/2018spring/abstractintepretation.pdf>
- [61] Lions, J.L., Luebeck, L., Fauquembergue, J.L., Kahn, G., Kubbat, W., Levedag, S., Mazzini, L., Merle, D., OHalloran, C.: Ariane 5 flight 501 failure report by the inquiry board (1996)
- [62] Lyman, M.: When and how to support static analysis tools with manual code review (2016), <https://www.synopsys.com/blogs/software-security/support-static-analysis-tools-with-manual-code-review/>
- [63] Macrae, C.: Vue. js: Up and Running: Building Accessible and Performant Web Apps. ” O’Reilly Media, Inc.” (2018)
- [64] Markovskiy, Y.: Range analysis with abstract interpretation. Semester Project, CS 263 (2002)
- [65] Milanova, A., Rountev, A., Ryder, B.G.: Precise call graphs for c programs with function pointers. Automated Software Engineering 11(1), 7–26 (2004)
- [66] Minz, O., Shomrat, M.: Constant propagation
- [67] Muchnick, S., et al.: Advanced compiler design implementation. Morgan kaufmann (1997)
- [68] Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer (2015)
- [69] Sagiv, P.M.: Abstract interpretation (2012), http://www.cs.tau.ac.il/~msagiv/courses/pa12-13/lital_sum.pdf
- [70] Salcianu, A.: Notes on abstract interpretation (2001)
- [71] Satav, S.K., Satpathy, S., Satao, K.: A comparative study and critical analysis of various integrated development environments of c, c++, and java languages for optimum development. Universal Journal of Applied Computer Science and Technology 1 (2011)
- [72] Smalltalk, R.: Csc 520 principles of programming languages (2005)
- [73] Stoltz, Gerlek, Wolfe: Extended ssa with factored use-def chains to support optimization and parallelism. In: 1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences. vol. 2, pp. 43–52 (1994)
- [74] Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. ACM Transactions on Programming Languages and Systems (TOPLAS) 13(2), 181–210 (1991)