TU WIEN Informatics

CodeDetective: Enabling isolated code execution

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Luise Ilg, MSc
Matrikelnummer 11775755

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc

Wien, 27. April 2025

Luise Ilg	Jürgen Cito

TU WIEN Informatics

CodeDetective: Enabling isolated code execution

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Software Engineering & Internet Computing

by

Luise Ilg, MSc
Registration Number 11775755

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc

Vienna, April 27, 2025

Luise Ilg	Jürgen Cito

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Luise Ilg, MSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 27. April 2025

Luise Ilg

Danksagung

Mit dieser Arbeit schließe ich ein bedeutendes und prägendes Kapitel meines akademischen Werdegangs ab. Mein Dank gilt all jenen, die mich nicht nur bei dieser Masterarbeit, sondern auch während meines gesamten Studiums begleitet und unterstützt haben.

Besonderer Dank gebührt Associate Prof. Dipl.-Ing. Dr.sc. Cito Jürgen, der mich im Rahmen dieser Arbeit mit seiner Expertise begleitet hat. Ich weiß es sehr zu schätzen, dass er sich selbst in arbeitsintensiven Phasen und während seines Urlaubs die Zeit nahm, meine Arbeit zu betreuen und mir hilfreiches Feedback zu geben. Auch in seinen Lehrveranstaltungen konnte ich viel Wissen mitnehmen, das mich über mein Studium hinaus begleiten wird.

Ich danke auch meinem Bruder, dessen Rückmeldungen und Diskussionen wertvolle Impulse für die Umsetzung und Evaluation des entwickelten Tools gegeben haben.

Ein großes Dankeschön gilt meiner Familie, die mich über den gesamten Studienverlauf hinweg begleitet, motiviert und unterstützt hat. Sie hat mir diesen Weg überhaupt erst ermöglicht.

Ebenso danke ich meinen Freundinnen und Freunden, die mir im Laufe des Studiums mit ihrer Unterstützung und ihrem Verständnis zur Seite standen.

Acknowledgements

With this thesis, I conclude a significant chapter of my academic journey. I would like to thank all those who supported me not only during the writing of this thesis, but also throughout my entire studies.

Special thanks go to Associate Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, who guided me through this work with his expertise. I deeply appreciate that he took the time to supervise my thesis and provide valuable feedback, even during busy periods and while on vacation. I also gained lasting insights from his courses, which will continue to accompany me beyond my academic path.

I would also like to thank my brother, whose feedback and discussions provided valuable input for the development and evaluation of the tool.

A big thank you goes to my family, who supported, motivated, and encouraged me throughout my studies. They made this journey possible in the first place.

Finally, I would like to thank my friends, who stood by me throughout my studies with their support and understanding.

Kurzfassung

In der Softwareentwicklung besteht häufig der Bedarf, gezielt bestimmte Codebereiche zu testen oder zu analysieren, ohne das gesamte Programm auszuführen. Dies wird jedoch schwierig, wenn der ausgewählte Bereich von den umgebenden Definitionen oder dem Kontext abhängt. In dieser Arbeit wird ein statisches Programm-Slicing-Verfahren vorgestellt, das die isolierte Ausführung von Codeausschnitten ermöglicht, die aus größeren Programmen extrahiert wurden. Die vorgestellte Methode identifiziert und rekonstruiert ausschließlich jene Teilmengen des ursprünglichen Codes, die erforderlich sind, um die Semantik eines ausgewählten Bereichs zu bewahren. Dadurch lässt sich das Ergebnis unabhängig vom ursprünglichen Kontext ausführen. Der Ansatz basiert auf einer rekursiven Abhängigkeitsanalyse durch Traversierung des abstrakten Syntaxbaums sowie des Aufrufgraphen. Dabei werden sowohl direkte als auch transitive Beziehungen zwischen Anweisungen aufgelöst. Ein Mechanismus zur Verfolgung von Variablenverwendungen unterscheidet zwischen auflösbaren und nicht auflösbaren Elementen. Letztere werden der Anwenderin oder dem Anwender über typgeprüfte Eingabeaufforderungen präsentiert. Die finale Teilmenge wird mittels topologischer Sortierung und strukturierter Rekonstruktion zu einem ausführbaren Skript umgebaut.

Der Ansatz wurde in dem Prototyp-Tool CodeDetective implementiert, das typisierte Python Eingabeprogramme unterstützt und anhand einer Benchmark-Suite mit zehn konzeptionellen Schwierigkeitsgraden evaluiert wurde. Die Evaluierung zeigt, dass das System eine Genauigkeit von über 82% erreicht und damit deutlich besser abschneidet als die Basis-Tools, denen es an Fähigkeiten zur Programmanalyse oder zur Verarbeitung von Benutzereingaben fehlt und lediglich etwa 10% bzw. 30% erreichten. Diese Ergebnisse verdeutlichen das Potenzial des Ansatzes, die isolierte Ausführung von Codeausschnitten zu ermöglichen, selbst wenn die relevanten Abhängigkeiten über den ausgewählten Bereich hinausgehen.

Abstract

In software development, developers often need to test or inspect specific regions of code without running the entire program. However, this becomes difficult when the selected fragment depends on surrounding definitions or context. This thesis presents a static program slicing approach for enabling the isolated execution of code snippets extracted from larger programs. The proposed method identifies and reconstructs only a subset of the original code that is required to preserve the semantics of a selected region, allowing the result to execute independently of its original context. The approach performs recursive dependency analysis by traversing the abstract syntax tree and call graph of the program, resolving both direct and transitive relationships between statements. A variable tracking mechanism distinguishes resolvable elements from unresolved ones, which are presented to the user through type-validated input prompts. The final subset is constructed into an executable script using topological sorting and structured rewriting.

The approach was implemented in the prototype tool CodeDetective, which supports typed Python input programs and evaluated on a benchmark suite spanning ten conceptual difficulty levels. The evaluation shows that the system achieves an accuracy score of over 82%, significantly outperforming baseline tools lacking program analysis or interactive input handling, which reached only approximately 10% and 30%. These results highlight the potential of the approach to enable isolated execution of code snippets, even when dependencies span beyond the selected region.

Contents

CHAPTER 1

Introduction

In modern software development, developers frequently interact with unfamiliar code, whether sourced from online platforms, legacy systems, large-scale projects, or even their own prior work. Understanding such code in context is often difficult, particularly when the surrounding logic, dependencies, or runtime behaviour is not immediately visible. This presents a significant challenge, as a deficiency of understanding may not only hinder efficient collaboration among developers but can also act as a barrier to comprehending the code thoroughly. Moreover, this lack of clarity increases the risk of introducing unintended bugs and errors into projects, further complicating the development process and potentially compromising the overall reliability of the software. Traditional approaches to comprehend unfamiliar code involve debugging, writing extensive tests, or copying and executing snippets in environments like Replit [1], an online interface for editing and running code directly in the browser across multiple languages.

However, each of these methods comes with its own set of drawbacks [24, 28, 10].

Copying and executing code snippets outside the project context using online platforms like Replit introduces complexities beyond just dependencies. It can be a cumbersome and error-prone process to execute only a subset of code in isolation. This involves manual tasks such as adjusting paths, declaring variables, modifying configurations, and ensuring consistent settings.

While debugging is the key to understanding code flow and identifying errors, it becomes increasingly challenging in large and nested projects. The complexity of nested functions, dependencies, and asynchronous operations make traditional debugging tools less effective, leading to long, laborious and cumbersome debugging sessions and increased cognitive load for developers. For instance, consider a scenario where a developer is unsure about the behaviour of a string operation and wants to inspect the following line of

[1] https://replit.com/

code `const maskedNumber = last4Digits.padStart(fullNumber.length, '*');` within a microservice architecture application. In order to examine this particular line, the developer might be compelled to initiate a debugging session. However, this requires a number of tasks, including starting connected services, compiling/building dependencies, ensuring the correct configuration of the entire project, and navigating through layers of code execution paths. Thus, initiating such a debugging session may be deemed disproportionately cumbersome given the substantial overhead involved, making it far less useful than the direct execution of the specific line under investigation.

In testing for code comprehension of a specific line, the effort associated with constructing a comprehensive test suite can be impractical, especially when the sole purpose is to quickly understand the operation of that line. This testing process lacks the agility required for quick and ephemeral exploration of code. For instance, in scenarios where developers need to comprehend a specific algorithm or logic in isolation, the overhead of writing an exhaustive test for that particular line can outweigh the benefits of quick comprehension. Consider again the example from above: a developer wants to investigate the basic string manipulation functionality of the line `const maskedNumber = last4Digits.padStart(fullNumber.length, '*');`. Now, imagine the idea of creating a thorough test suite for this individual line. This would involve developing a detailed set of test cases to thoroughly check how the code behaves in various scenarios. However, given the simplicity of the operation, which involves padding the final four digits of a number with asterisks, an extensive test suite might be considered overly extensive.

To address this challenge, the thesis proposes a slicing-based approach for isolating and executing selected code regions. Given a user-specified range of source lines, the system analyses the program to determine all required dependencies. Variables that are referenced but not defined within the selected region are treated as unresolved and resolved through interactive user input, ensuring that the resulting code is executable. The system then builds a self-contained program from the selected region and executes it automatically, producing observable output based on the provided inputs.

This approach is realized in the form of a prototype implementation named CodeDetective, targeting statically typed Python projects. The system integrates program analysis, variable tracking, user interaction, and code reconstruction in a fully automated workflow. Developers can specify a line or range of lines in a Python source file and the system will extract and execute the corresponding slice, returning its result while preserving all necessary dependencies and semantics.

The central research questions explored in this thesis are:

- **RQ1** How can we build a program analyser that produces a subset of executable code snippets from a source file?
- **RQ2** How well does the developed tool perform in evaluating code snippets of varying complexity and structure within controlled testing environments?

The thesis begins with an overview of related work in Chapter 2, followed by background on program representation and slicing in Chapter 3. Chapter 4 introduces the general approach for isolating and executing code regions, while Chapter 5 describes the concrete implementation of this approach in the form of the prototype tool CodeDetective. Chapter 6 outlines the evaluation methodology, and Chapter 7 presents and discusses the results. The thesis concludes in Chapter 8 with a summary of findings and directions for future work.

CHAPTER 2

Related Work

Execution of individual code snippets have been investigated, among others, by Godefroid in the context of software testing. In his paper "Micro execution" [11], Godefroid introduces MicroX, a novel approach to unit testing by enabling the execution of isolated code fragments without the need for a traditional test setup, such as a test driver or predefined input data. It achieves this by compiling code into x86 assembly instructions and executing them one by one, capturing all memory operations and intervening as necessary.

Code Runner [20] is a Visual Studio Code extension that enables developers to execute selected code snippets or entire code files in various programming languages, including JavaScript, Python, and C++. While Code Runner facilitates code execution, it lacks advanced program analysis capabilities. Specifically, it does not resolve dependencies within the selected code, instead merely copying the code snippet into a new temporary file for execution. This approach overlooks critical aspects of code comprehension and execution, potentially leading to incomplete or inaccurate results. For instance, without addressing dependencies, programs executed using Code Runner may fail or produce unintended outcomes. Figure 2.1 illustrates an example of Code Runner in action.

Souza and Pradel [27] address the ongoing challenge of executing incomplete code, including code snippets sourced from internet platforms or within large projects. They introduce LExecutor, a learning-guided technique, in order to execute arbitrary code snippets in an underconstrained way. The focus lies on preventing execution crashes due to missing data. Thus, LExecutor uses a neural model to estimate appropriate values in situations when the program would typically crash.

To assist with code comprehension during runtime, Lerner introduced Projection Boxes, an innovative visualization technique designed to provide continual feedback on runtime values within a Live Programming (LP) environment [22]. By allowing programmers to selectively visualize subsets of program semantics, Projection Boxes mitigate information

Figure 2.1: **Code Runner example** - Original code file (`main.py`) containing the selected code line 3 (`print(x)`) alongside the temporary file `tempCodeRunnerFile.py` which was automatically generated by Code Runner. The `tempCodeRunnerFile.py` demonstrates that only the selected code was copied into a temporary file for execution, without resolving dependencies within the selected code.

overload while accommodating individual user preferences. The study conducted in the course of the paper indicates that Projection Boxes are effective for identifying and understanding special operations, as well as for locating and fixing bugs.

In their paper 'Validating AI-Generated Code with Live Programming,' Ferdowsi et al. introduce LEAP (Live Exploration of AI-Generated Programs), a Python environment that integrates live programming with an AI assistant to validate code snippets generated by AI tools [8]. The study investigates how well LP addresses the challenge of validating AI suggestions by reducing over- and under-reliance on AI-generated code. By utilizing Projection Boxes as part of the LP environment, the authors demonstrate how LP lowers the cost of validation by execution, making it more feasible for developers to validate AI suggestions.

In a recent study [34], the concept of an instructive copilot as a programming assistant has been introduced, aiming to provide instant explanations of generated code. The tool Ivie (instantly visible in-situ explanations) offers concise AI-generated explanations of newly generated code segments, assisting programmers in grasping unfamiliar APIs and coding patterns present in the generated code. By seamlessly integrating with programming assistants like GitHub Copilot, Ivie enhances code comprehension without adding significant distraction or task load.

SpotFlow [16] introduces a novel paradigm in program analysis by facilitating the extraction of runtime insights, including method calls, executed lines, variable states, and exceptions, thereby enhancing code comprehension and debugging processes. By operating at the method level and leveraging the sys.settrace function [4] and the inspect module [3] from Python, SpotFlow enables fine-grained analysis, revolutionizing dynamic program analysis for Python developers.

Frama-C [2] is a powerful tool for program analysis, providing a suite of static analysis techniques for C and experimental C++ code. Its collaborative framework and over 27

plug-ins empower software engineers with correct bug detection, functional specification manipulation, and code adherence verification [7]. Focused on correctness and automation, the combination of value analysis and slicing capabilities in Frama-C enables in-depth code analysis, ensuring comprehensive understanding and assurance of software systems [21].

While previous research has explored isolated code execution and comprehension, this thesis approaches this challenge from a different perspective. Unlike existing methodologies such as MicroX, which focuses on executing code at the compiled assembly level, or LExecutor, which addresses incomplete code execution by estimating missing values, our work is designed for higher-level languages and emphasizes user interaction for providing input values. Whereas Code Runner offers basic code execution functionalities without considering dependencies or providing advanced program analysis features, our tool incorporates program analysis to resolve dependencies and provide a more comprehensive understanding of code behavior. While Frama-C and SpotFlow excel in program analysis, particularly in static and dynamic analysis techniques, their approach diverges from ours. They emphasize comprehensive code analysis across entire programs, whereas our tool uniquely enables developers to isolate and execute specific portions of code, offering targeted execution capabilities for focused analysis. Additionally, while tools like Projection Boxes offer valuable capabilities within live programming environments, our focus is on enabling developers to execute specific portions of code outside of such environments. Unlike Ivie, which requires integration with specific programming assistants and operates within specific IDEs like Visual Studio Code, our tool aims to be a standalone solution, providing flexibility for developers across various development environments. Additionally, while Ivie focuses solely on explaining code generated from programming assistants, our tool targets a broader scope by enabling developers to execute specific portions of code, regardless of their origin or generation method. With the development of CodeDetective, our research aims to provide a specialized solution for code exploration and comprehension in scenarios where users need to investigate and understand individual code snippets efficiently and effectively.

CHAPTER 3

Background Information

This chapter provides an overview of fundamental concepts necessary for understanding the proposed approach. It introduces different ways to represent source code, as well as techniques for analysing and extracting relevant parts of a program.

Section 3.1 covers key program representations, including Abstract Syntax Tree (AST), Dependency Graphs and Call Graphs, which are essential for structuring and analysing source code. Section 3.2 introduces program slicing, a technique for isolating specific parts of a program while preserving its intended behaviour.

3.1 Program Representation

Representing the source code is essential for various software engineering tasks such as code classification, code clone detection or method name prediction. To facilitate these tasks, different models are used to capture the syntactic and semantic structure of a program. Among the most widely used representations are the Abstract Syntax Tree (AST), Dependency Graph, and Call Graph, each serving a distinct role in analysing software system[29]. The following subsections explain each of those models in more detail. Subsection 3.1.1 provides more insights on AST, while subsection 3.1.2 discusses dependency graphs and 3.1.3 focuses on call graphs.

3.1.1 AST - Abstract Syntax Tree

The Abstract Syntax Tree (AST) serves as a structured model of source code, capturing its logical structure in a tree format. It is widely utilized in compilers, interpreters, and automatic code generation tools. Each node signifies a different programming element, such as expressions, statements, or declarations, while edges illustrate the relationships between those elements. At the top of the tree is the root node, which serves as the central entry point and typically corresponds to the entire program or the main construct

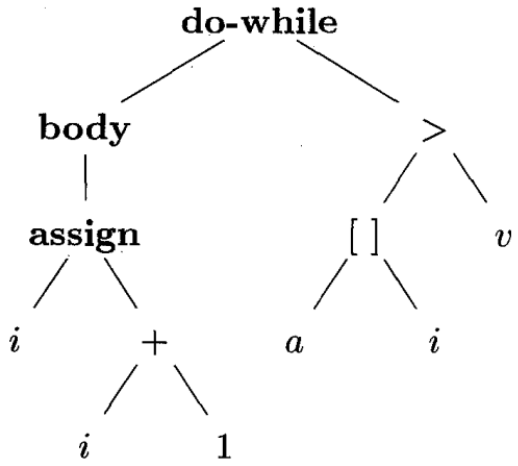

Figure 3.1: **Abstract Syntax Tree** - Illustration of an Abstract Syntax Tree representation of expression `do i = i + 1; while(a[i]>v);` [6].

being analysed. Branching from the root are child nodes, each representing smaller code components, such as expressions, statements, or function calls. These child nodes may further branch into more specific elements, forming a nested structure that mirrors the logical flow of the program. At the lowest level, the leaf nodes serve as terminal elements, representing fundamental components such as variables, constants, or operators [6, 15, 14]. Figure 3.1 illustrates an AST of code expression `do i = i + 1; while(a[i]<v);`. The root is the `do-while` construct. The left subtree represents the loop `body`. The body contains an `assignment` operation, where i is updated with the result of $i + 1$. This is structured hierarchically, with the assignment operation breaking down into two components: the left-hand side, which is the variable i, and the right-hand side, which is an addition operation $+$. The addition itself further decomposes into its operands, the variable i and the constant 1, forming a nested representation of the arithmetic expression. The right subtree of the root node represents the loop condition, expressed as a greater-than comparison ($>$), which checks whether an array element $a[i]$ is greater than v. This comparison consists of two parts: the left-hand side, which is an array access operation [] retrieving the value stored at index i of array a, and the right-hand side, which is simply the variable v.

The Python Abstract Syntax Tree [1] is a specific implementation of the general AST concept, designed for the Python programming language. Like other ASTs, it represents the syntactic structure of Python code as a tree, where each node corresponds to a syntactic construct such as expressions, statements, or functions. The Python Abstract Syntax Tree is generated using the built-in `ast` module, which parses Python source code into a tree structure. This tree can then be analysed, transformed, or optimized, allowing developers to inspect and manipulate Python code programmatically. Through

the `ast` module, it is possible to traverse the tree, modify individual nodes, or extract information to facilitate tasks like static analysis or code refactoring.

3.1.2 Dependency Graph

A dependency graph is a directed graph that models the dependencies between various elements in a system. These graphs are widely used in software engineering, database systems, project management, and computational sciences, where they help visualize relationships that dictate execution order, computation flow, or resource dependencies. Each node in a dependency graph represents an entity, such as a function, module, variable, or task, while directed edges indicate that one entity depends on another [6, 9, 17].

The structure of a dependency graph is defined by different types of nodes. At the highest level, root nodes represent independent entities that do not rely on any other nodes, serving as the starting points in the dependency chain. From the root nodes, dependencies propagate through intermediate nodes, which have both incoming and outgoing edges, meaning they depend on some nodes while also serving as dependencies for others. At the lowest level, leaf nodes have no outgoing edges, meaning they do not influence any other nodes, but instead serve as final elements in the chain. The edges connecting these nodes represent dependency relationships and determine the order in which elements must be processed. A directed edge from node `A` to node `B` indicates that `A` depends on `B`, implying that `B` must be resolved, computed, or executed before `A`.

Dependency graphs are widely used in software compilation, where they help determine the order in which files or modules should be compiled. For example, if a program consists of multiple source files that include one another, the compiler must first resolve files that do not depend on any others before compiling dependent files [6].

Dependency Resolution

In dependency graphs, the relationships between elements dictate the order in which they must be processed. However, resolving these dependencies in the wrong sequence can lead to errors or inefficiencies. For example, in software compilation, a file that depends on another must not be compiled before its dependency. To address this issue, topological sorting is used to establish a valid processing order for the elements in a dependency graph. A topological sort produces a linear ordering of nodes in a direct acyclic graph such that for every directed edge from node `A` to node `B`, `A` appears before `B` in the order. This guarantees that dependencies are always resolved before the elements that rely on them [6].

One common way to perform a toplogical sort is using Depth-first search (DFS). The DFS-based approach ensures that each node is added to the final order only after all of its dependencies have been explored. The algorithm does the following [18, 5]:

1. **Traversal and Marking**: The algorithm iterates over all nodes in the graph, maintaining a set of visited nodes to track progress. When a node is encountered for the first time, it is marked as visited.

2. **Recursive Exploration**: If the node has outgoing edges (i.e., dependencies), the algorithm recursively visits each of the connected nodes before proceeding further. This ensures that all dependencies are processed before the current node.

3. **Post-Processing and Ordering**: Once all dependencies of a node have been visited, the node itself is added to a stack or a list that records the topological order. This ensures that elements are added only after their dependencies have been fully processed.

4. **Final Ordering**: After all nodes have been processed, the stack (which maintains a reverse order) is reversed to obtain the correct topological ordering.

Algorithm 3.1 shows the pseudocode implementation of topological sort algorithm.

Algorithm 3.1: Pseudocode to perform topological sort using Depth-first search (DFS) [18]

```
Input: A directed acyclic graph (DAG) G = (V, E)
Output: A topological ordering of the vertices in V
Initialize an empty stack S;
Initialize an empty set visited;
Function DFS(v):
    if v ∈ visited then
        return
    end
    Mark v as visited;
    for each neighbor u of v do
        if u ∉ visited then
            DFS(u);
        end
    end
    Push v onto stack S;
for each vertex v ∈ V do
    if v ∉ visited then
        DFS(v);
    end
end
return stack S (in reversed order);
```

3.1.3 Call Graph

A call graph is a directed graph that represents the calling relationships between functions or methods in a program. Each node in the graph corresponds to a function, and a directed edge from one node to another indicates that the function represented by the source node calls the function represented by the destination node. Call graphs are essential tools for analysing program flow and understanding the structure of code. They are widely used in fields like static analysis, optimization, and program comprehension as they provide a clear visualization of how functions interact and depend on each other [23, 26, 31, 13, 12].

In a typical call graph, the root node represents the entry point of a program (e.g., the `main` function), while its child nodes represent functions that are invoked within the program. These edges can also capture recursive relationships, where a function calls itself, helping developers detect infinite recursion, dead code, or performance inefficiencies [31].

Figure 3.2 illustrates an example of a call graph for a simple Java program. In this example, the `main` function serves as the entry node, meaning it is the starting point of execution when the program runs. From `main`, two functions are invoked: `A` and `B`, creating two outgoing edges in the graph. Function `A` subsequently calls `C`, meaning any execution that reaches `A` will also lead to the execution of `C`. Function `B`, on the other hand, contains a recursive call, invoking itself within its own execution. This is represented by a looped edge in the graph. The call graph visually represents these relationships, showing how functions interact during program execution.

```
class Main {
  public static void main(String[] args) {
    A();
    B();
  }
  public static int A(){
    C();
  }
  public static void B(){
      B();
  }
}
```

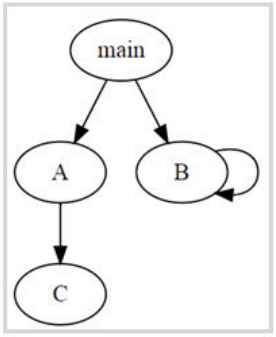

Figure 3.2: **Call Graph** - A call graph representing function calls in a Java program [31]

.

Practical Python Call Graphs (PyCG) [26] is a static analysis tool designed to generate call graphs for Python code. It constructs these graphs by analysing function calls without executing the program, making it valuable for code analysis and dependency

```
(1)   read(n);
(2)   i := 1;
(3)   sum := 0;
(4)   product := 1;
(5)   while i <= n do
      begin
(6)     sum := sum + i;
(7)     product := product * i;
(8)     i := i + 1
      end;
(9)   write(sum);
(10)  write(product)
```

(a)

```
read(n);
i := 1;

product := 1;
while i <= n do
begin

  product := product * i;
  i := i + 1
end;

write(product)
```

(b)

Figure 3.3: **Program Slicing example** - **(a)** The original program computes both sum and product of the first `n` natural numbers. **(b)** The sliced program retains only the computations relevant to the final value of `product`, removing unnecessary operations. [30]

tracking. PyCG is particularly useful for exploring large codebases, identifying function dependencies, and performing various types of static analysis. By using PyCG, developers can visualize function interactions and gain deeper insights into the structure of their Python applications.

3.2 Program Slicing

Program slicing is a technique in software engineering that extracts relevant parts of a program based on a given criterion. Weiser [32] first introduced program slicing in 1981 as a method to simplify code while maintaining its intended functionality.

A *slice* is a reduced version of the original program that still produces the same results for a specific subset of behaviour. Even though it is smaller, the slice remains executable on its own and preserves the intended functionality of the extracted portion [32, 33, 30].

Figure 3.3 illustrates how program slicing simplifies code by discarding irrelevant computations while maintaining accuracy for the selected variable [30]. The original program shown in Figure 3.3 (a) reads an integer `n` and calculates both the `sum` and the `product` of the first `n` natural numbers. The program initializes two variables, `sum` and `product`, and iterates from `1` to `n`, updating these values within a loop. Once the loop completes, it prints both computed results. In contrast, Figure3.3 (b) displays a sliced version of this program, where only the computations affecting `product` are preserved. The slicing is performed with respect to the criterion `(10, product)`, which indicates that we

are analysing the value of `product` at line `10`. As a result, all computations related to `sum` have been removed, as they do not contribute to the final value of `product` [30]. This slicing is performed based, on the criterion `(10, product)`, meaning the analysis focuses on how `product` is computed at line `10`.

There are two primary types of program slicing: *static slicing* and *dynamic slicing*. Static slicing is performed without considering specific program inputs, making it useful for general program analysis, security auditing, and software maintenance. Dynamic slicing, on the other hand, focuses on a specific execution of the program by analysing how statements interact under a given input. Unlike static slicing, which considers all possible execution paths, dynamic slicing only includes the statements that actually affect the computation for that particular input. This makes it especially useful for debugging, as it helps developers understand how a specific result was produced without analysing unnecessary parts of the code [33, 30].

Several techniques have been developed to perform program slicing efficiently. One of the most widely used approaches is *Program Dependence Graph (PDG)*-based slicing, where dependencies between program statements are explicitly modelled to enable accurate slice extraction. PDG-based slicing captures both data dependencies (tracking variable definitions and their uses) and control dependencies (tracking execution flow based on conditions), making it a precise and effective method for analysing program behaviour [25, 17, 33].

Program slicing can also be categorized into *backward slicing* and *forward slicing*. Backward slicing identifies all statements that contribute to the value of a particular variable at a given program point, making it particularly useful for debugging and impact analysis, as it helps track the origin of a value by tracing dependencies back through the code. Forward slicing, on the other hand, determines all statements affected by a specific variable or computation, which is often used for program understanding and security analysis, as it helps assess how changes to a variable propagate through a program[33, 30].

The applications of program slicing are extensive. In software debugging, slicing helps developers isolate faulty code sections, reducing the time needed to locate and fix errors [25, 30]. In performance optimization, it allows compilers to eliminate redundant or dead code while maintaining correctness [33, 30]. Furthermore, slicing enhances software maintenance and comprehension, enabling developers to extract and analyse only the relevant parts of a large and complex codebase [30].

CHAPTER 4

Approach

This chapter describes the general approach for isolating and executing selected regions of code within a larger software project. The objective is to enable precise, dependency-aware program slicing that extracts only the code relevant to a specific region under investigation, while ensuring correctness and executability.

Given a program P consisting of multiple source files, the user specifies a target range of lines, denoted as the region under investigation $RUI \subseteq P$. The approach analyses the program to identify all statements required to execute RUI in isolation. Variables referenced within RUI that are not defined within the selected range are treated as input parameters and resolved by binding them to user-provided values. The process produces a standalone executable slice $P_S \subseteq P$ that captures only the subset of code necessary for executing the selected region, including reconstructed dependencies and runtime inputs. This slice is then executed automatically and the resulting output is presented to the user

This process is divided into five conceptual phases:

1. Setup: Capture the selected region and prepare the program for analysis
2. Program Analysis: Determine all dependencies relevant to RUI
3. User Input Collection: Prompt for unresolved variable values and validate user input
4. Code Generation: Reconstruct a minimal standalone program from the extracted slice
5. Execution: Run the generated program to observe the behaviour of RUI

Figure 4.1 provides a visual overview of the overall workflow of the proposed approach. It highlights the key phases within the system, the role of user interaction and the

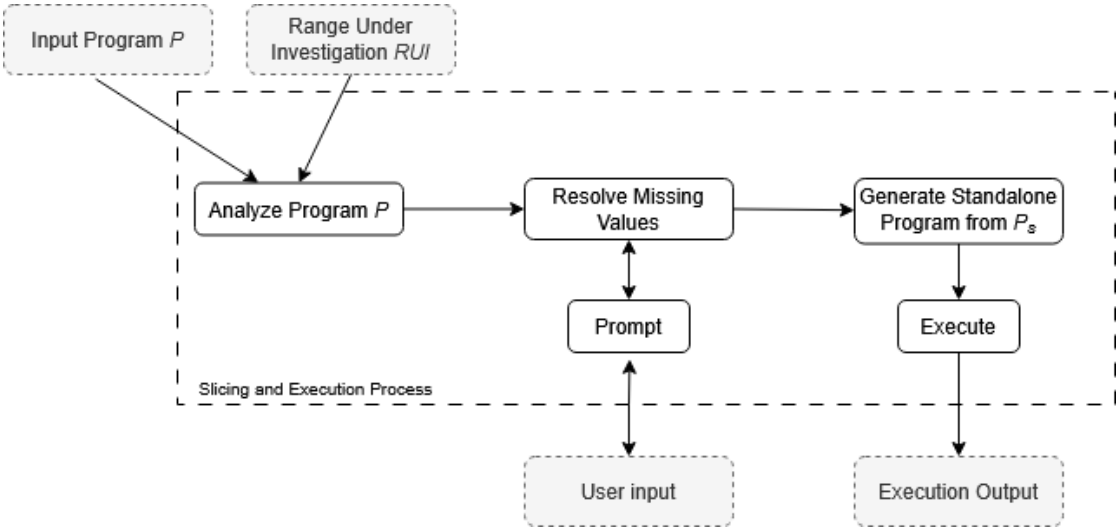

Figure 4.1: Overview of the overall workflow of the proposed approach. The process begins with code selection and analysis, followed by dependency resolution, user input collection, code generation and execution. External entities are shown as dotted boxes.

separation between internal processes and external entities such as the input program, user input, and final output.

To support the explanation of the approach, the following two running examples are introduced and referenced throughout the following sections.

Simple Example: This first example consists of two statements where a variable is assigned and subsequently printed after being transformed through a string operation. This minimal snippet demonstrates basic slicing and variable usage

```
1 message: str = "hello"
2 print(message.upper())
```

Listing 4.1: Simple Example

The region under investigation consists of line 2. The variable `message` is not defined within this region, and therefore must be resolved as part of the slicing process by requesting a value from the user.

Complex Example: The second example illustrates a more complex scenario, emphasizing interprocedural and cross-file dependencies. It involves user-defined classes, function invocations, and import statements, with logic distributed across two separate modules. This setup highlights how the tool resolves dependencies that span across functions and files, offering a more realistic example of program slicing in practice.

```
1 class User:
2     def __init__(self, name: str, is_admin: bool):
3         self.name = name
```

```
4         self.is_admin = is_admin
5
6     def has_admin_rights(self) -> bool:
7         return self.is_admin
8
9     def __str__(self) -> str:
10        return self.name
```

Listing 4.2: Complex example - File: `user.py`

```
1 from user import User
2
3 def log_access(user: User) -> None:
4     print(f"Access denied for: {user}")
5
6 def process_user(user: User) -> None:
7     if user.has_admin_rights():
8         print("Access granted")
9     else:
10        log_access(user)
11    print("processed user")
12
13 def log_access_attempt() -> None:
14    print("User attempted access")
```

Listing 4.3: Complex example - File: `access_control.py`

The Range Under Investigation in this example includes lines 7 to 10 within the `process_user` function in the `access_control.py` file. The analysis must identify the dependency on the `User` class defined in `user.py`, as well as the `log_access` function declared on line 3. In addition, the variable `user` is not defined within the selected code and must be supplied by the user at runtime.

4.1 Setup Phase

The slicing process begins with the identification of a region of interest and the preparation of the program for structural analysis.

Let P denote the full program, organized as a finite set of source files $F = \{f_1, f_2, \ldots, f_n\}$. The user provides a file $f_{\text{rui}} \in F$ along with a continuous interval of line numbers $l_s \ldots l_e \subseteq \texttt{Lines}(f_{\text{rui}})$, which defines the region under investigation. This region is formalized as:

$$RUI = \{s \in f_{\text{rui}} \mid \texttt{line}(s) \in [l_s, l_e]\}$$

where s ranges over the statements in the specified file.

During setup, a call graph is constructed to support interprocedural dependency resolution. Let

$$G_{CG} = (V_{CG}, E_{CG})$$

denote the call graph, where each vertex $v \in V_{CG}$ corresponds to a function or method, and each edge $(v_i, v_j) \in E_{CG}$ represents a potential call from v_i to v_j.

This setup phase establishes the foundation for precise dependency analysis in subsequent steps. The extracted region RUI, the program P and the call graph G_{CG} serve as input to the subsequent analysis phase.

4.2 Program Analysis

The Program Analysis phase is responsible for analysing the input program using structured representations such as the Abstract Syntax Tree and the Call Graph. This phase employs a static program slicing technique via a Dependency Graph (DG) to extract relevant portions of code based on dependencies. The primary goal of program analysis is to systematically track variable usage, infer types, and identify dependencies between statements to later construct a minimal executable slice of the statements within the RUI.

To achieve this, the input program is first parsed into an AST representation, which provides a structured view of the syntax of the program. This representation serves as the foundation for the slicing process.

The analysis method follows the static program slicing approach, implemented through a Dependency Graph (DG). The dependencies associated with each statement are identified and resolved recursively to build the slice. Additionally, a Variable Resolution Tracker (VRT) is used to determine missing values necessary for execution. The VRT distinguishes between resolved and unresolved variables:

- **Resolved Variables**: Variables that are assigned within the slice.
- **Unresolved Variables (User Variables)**: Variables that are used within the slice but not defined within RUI. These require user input for execution.

Algorithm 4.1 outlines the key steps involved in the program analysis process. The first step is to initialize the Dependency Slicer S, which serves as the core component of the slicing mechanism. During this initialization, among others, an empty Dependency Graph DG and a Variable Resolution Tracker VRT are created. The DG is responsible for capturing the dependencies between program statements, while the VRT tracks variables used within the slice, distinguishing between those that are already resolved and those that require user input.

Once the slicer is initialized, the second step involves iterating over each statement $stmt_c$ within the RUI. Each statement is passed to the `visit` function `visit(`$stmt_c$`)`, which determines how the statement should be processed. Within this function, the statement type $stmt_t$ is identified and the corresponding handler H is instantiated. The handler is responsible for processing $stmt_c$ according to its syntactic role in the program.

Algorithm 4.1: Program Analysis Process

Input: Input Program P, RUI: single line l or range $[l_s, l_e]$
Output: Dependency Graph DG and Variable Resolution Tracker VRT

```
/* Step 1: Initialize the Dependency Slicer                    */
```
Initialize Dependency Slicer S with empty DG and VRT ;

```
/* Step 2: Traverse the Statements in the Range under Investigation
   (RUI)                                                       */
```
for *each statement* $stmt_c$ *in* RUI **do**
  `visit(`$stmt_c$`)` ;
end

```
/* Step 3: Recursive Analysis of Each Statement                */
```
Function `visit(`$stmt_c$`)`**:**
  Identify statement type $stmt_t$;
  Retrieve appropriate handler H based on $stmt_t$;
  H`.handle(`$stmt_c$`)` ;

; `// Defined within handler class` H`:`
Function `handle(`$stmt_c$`)`**:**
  Identify used variables V_u in $stmt_c$;
  for *each variable* $v \in V_u$ **do**
    Add v to VRT, marked as *resolved* or *unresolved* ;
  end
  Identify dependencies D of $stmt_c$;
  for *each dependency* $d \in D$ **do**
    Locate the previous statement $stmt_p$ d occurs in;
    Add $stmt_p$ as a node in DG and link it to $stmt_c$;
    `visit(`$stmt_p$`)` ;
  end

return (DG, VRT) ;

The third step is the core of the slicing process, in which the handler processes the statement $stmt_c$ by extracting its relevant variables and dependencies. The handler first identifies all variables used in the statement $stmt_c$ and updates the VRT accordingly. Variables that are assigned a value within $stmt_c$ and $stmt_c$ is within RUI, are marked as *resolved*, while those that are used but not assigned within $stmt_c$ or defined outside of RUI remain *unresolved* and are classified as user variables that require external input. Next, the handler identifies dependencies by determining which previous statements $stmt_p$ contribute to the execution of $stmt_c$. Each relevant previous statement $stmt_p$ is added to the DG, and the `visit` function is called recursively to ensure that all dependencies are traced back to their origins.

Through this recursive resolution of dependencies, the DG captures all statements required to reconstruct an executable slice, while VRT identifies any unresolved values that must be supplied by the user.

4.2.1 Initialization of the Dependency Slicer

Before the program analysis can begin, the Dependency Slicer S must be properly initialized. The initialization phase sets up essential components, including the Dependency Graph DG, Variable Resolution Tracker VRT, visited set, and recursion stack. Each of these components serves a distinct purpose, ensuring that the slicer can correctly track dependencies, resolve variable usages, and efficiently navigate the control flow of the input program R_T.

The procedure for initializing S involves the following steps:

1. Parse the source code into an Abstract Syntax Tree representation.
2. Normalize and transform import statements within the AST for accurate dependency tracking.
3. Initialize the Dependency Graph DG and Variable Resolution Tracker VRT.
4. Set up the visited set and recursion stack for efficient traversal.

The remainder of this section outlines the initialization process in detail, covering both preprocessing steps and the setup of internal data structures used by the slicer.

AST Preprocessing and Import Normalization

Before the slicing process begins, the input program is parsed into an Abstract Syntax Tree, a structured representation of the source code. Each statement in the program becomes an AST node, including import statements, which require normalization to ensure analysability.

To facilitate accurate dependency analysis, import statements are normalized by applying a transformation function T over the set I of import nodes extracted from the AST. This

transformation rewrites each import into a canonical form by splitting compound imports into separate statements, resolving relative imports into absolute imports, expanding wildcard imports into explicit name-based imports and ensuring that every imported symbol has an explicit alias, defaulting to the original name if none was provided.

Formally, the transformation function $T : I \to I'$ is defined as follows:

$$T(i) = \begin{cases} \{\, \texttt{import}\ n\ \texttt{as}\ a \,\} & \text{if } i = \texttt{import}\ n \text{ or } i = \texttt{import}\ n\ \texttt{as}\ a \\ \{\, \texttt{import}\ n_j\ \texttt{as}\ n_j \mid j = 1, \ldots, k \,\} & \text{if } i = \texttt{import}\ n_1, \ldots, n_k \\ \{\, \texttt{import}\ n_j\ \texttt{as}\ a_j \mid j = 1, \ldots, k \,\} & \text{if } i = \texttt{import}\ n_1\ \texttt{as}\ a_1, \ldots, n_k\ \texttt{as}\ a_k \\ \{\, \texttt{from}\ x\ \texttt{import}\ n_j\ \texttt{as}\ n_j \mid j = 1, \ldots, k \,\} & \text{if } i = \texttt{from}\ x\ \texttt{import}\ n_1, \ldots, n_k \\ \{\, \texttt{from}\ x\ \texttt{import}\ n_j\ \texttt{as}\ a_j \mid j = 1, \ldots, k \,\} & \text{if } i = \texttt{from}\ x\ \texttt{import}\ n_1\ \texttt{as}\ a_1, \ldots, n_k\ \texttt{as}\ a_k \\ \{\, \texttt{from}\ x\ \texttt{import}\ s\ \texttt{as}\ s \mid s \in \text{symbols}(x) \,\} & \text{if } i = \texttt{from}\ x\ \texttt{import}\ * \\ \{\, \texttt{from}\ \text{resolve}(y)\ \texttt{import}\ n_j\ \texttt{as}\ a_j \mid j = 1, \ldots, k \,\} & \text{if } i = \texttt{from}\ .y\ \texttt{import}\ n_1, \ldots, n_k \end{cases}$$

where symbols(x) is the set of explicitly importable names defined in module x and resolve(y) represents the absolute path resolution for a relative import.

Specifically, the transformations handle the following cases:

- A simple import such as `import os` is normalized to `import os as os`.
- Compound imports without explicit aliases are split into individual statements. For example, `import os, sys` transforms into `import os as os` and `import sys as sys`.
- Compound imports with explicit aliases, such as `import numpy as np, pandas as pd`, are preserved with aliases, but transformed into individual statements.
- Wildcard imports, such as `from module import` $*$, are expanded into explicit individual statements for each name $s \in$ symbols($module$).
- Relative imports, like `from .submodule import function`, are converted into absolute imports, for instance `from package.submodule import function`, where resolve($\cdot$) denotes the resolution of relative paths to absolute module paths.

This normalization simplifies subsequent dependency analysis by ensuring that each imported symbol is clearly and individually represented in the AST.

Dependency Graph *DG*

The Dependency Graph *DG* is a directed graph used to model the dependencies between program statements. It is formally defined as:

$$DG = (V_{DG}, E_{DG})$$

where V_{DG} is the set of vertices, each corresponding to a statement in the AST, i.e., $v \in V_{DG}$ represents a program statement. Each edge $e \in E_{DG}$ is formally defined as a triplet $(v1, v2, \sigma)$, where $v1$, $v2 \in V_{DG}$ and σ is a symbol that describes the anchor of the dependency. The vertex $v1$ corresponds to the source statement (i.e., the statement that a dependency originates from), and $v2$ is the target statement (i.e., the statement the source vertex $v1$ depends on). Hence, the edge e indicates that v_1 depends on v_2, meaning that the execution or semantics of v_1 relies on information defined or established by v_2. The symbol σ expresses why the dependency exists. It identifies the relevant construct that links the two statements. For example, if a variable is annotated with a custom class type, an edge may exist from the `AnnAssign` node to the corresponding `ClassDef`, with σ being the name of that class. Similarly, if a function `f` calls another function `g`, there would be an edge from the body of `f` to the `FunctionDef` node of `g`, labeled with $\sigma =$ "g".

Variable Resolution Tracker VRT

The Variable Resolution Tracker VRT is a data structure used to track the resolution states of variables throughout the program. It keeps track of whether a variable has been defined or used, and whether its value is resolved or unresolved. Formally, the VRT is defined as a tuple

$$VRT = (R, U)$$

where R is the set of *resolved variables* and U is the set of *unresolved variables*. At the time of initialization, both sets are empty:

$$R = \{\} \quad \text{and} \quad U = \{\}.$$

A variable is considered *resolved* if it is assigned a value within the slice, meaning it is defined in every execution path of the RUI. Conversely, a variable is marked as *unresolved* if it is used within the slice but not defined therein, indicating that its value must be provided externally. This distinction is fundamental for identifying variables that require external input for the correct execution of the final slice. During the slicing process, the VRT is dynamically updated to record variable definitions and usages as they are encountered.

Visited Set and Recursion Stack

To efficiently traverse the AST statements relevant to the RUI and avoid redundant processing, the Dependency Slicer employs two supporting data structures: the visted set $VisitedSet$ and the recursion stack $RecursionStack$.

The visited $VisitedSet$ set is defined as:

$$VisitedSet \subseteq V_{AST},$$

where V_{AST} is the set of all nodes (i.e., statements) in the AST. At initialization, the visited set is empty:

$$VisitedSet = \{\}.$$

This set is used to record nodes that have already been processed, ensuring that each node is visited only once during the analysis.

The recursion stack $RecursionStack$ is a data structure that tracks the current chain of recursive invocations during the AST traversal. It is formally considered as an ordered list:

$$RecursionStack = [\,],$$

where the empty list indicates that no nodes are currently being processed. As the slicer S traverses nested structures (such as functions or control-flow blocks), nodes are pushed onto the recursion stack and subsequently popped once processing is complete. This mechanism helps prevent infinite recursion and enables correct backtracking during the dependency analysis.

Together, the visited set and recursion stack ensure that the slicer can efficiently and correctly traverse the AST without redundant processing or cycles.

4.2.2 Slicing Traversal and Dependency Resolution

Once the Dependency Slicer S has been initialized, the slicing process begins by identifying and resolving all dependencies relevant to the selected RUI. This process aims to construct a minimal executable slice $P_S \subseteq P$, where P is the original program and P_S contains only the statements necessary for the semantically correct execution of the selected range.

To compute this slice, the slicer recursively traverses the AST and determines which program statements must be included. The traversal is managed through recursive invocations of a function `visit`, which serves as the core of the slicing mechanism.

To ensure correctness and avoid redundant processing, the slicer maintains two key sets throughout the traversal:

- The visited set $VisitedSet \subseteq V_{AST}$, as defined in Section 4.2.1, which records all statements that have already been processed.
- The recursion stack $RecursionStack = [\,v_1, \ldots, v_m\,]$, a list tracking the current recursive path during the traversal.

The traversal is initiated on each statement $s \in RUI$. However, the process is not limited to statements within the RUI, as resolving dependencies often requires analysing

additional statements $s' \in V_{AST} \setminus RUI$ that influence the execution of the RUI. Each such statement is passed to the `visit` function, which operates as shown in Algorithm 4.2

Algorithm 4.2: Recursive Visit Function

Input: Statement $s \in V_{AST}$, current Dependency Slicer $S = (DG, VRT, \dots)$

```
if s ∈ VisitedSet or s ∈ RecursionStack then
    return
end
Push s onto RecursionStack ;
Mark s as visited: VisitedSet ← VisitedSet ∪ {s} ;
Determine the type of s and dispatch to the corresponding handler H ;
H analyses s, identifies dependencies D_s = {s_1, ..., s_k} ;
foreach s_i ∈ D_s do
    Add edge (s, s_i, σ_i) to DG ;
    visit(s_i) ;                                   // Recursive call
end
Pop s from RecursionStack ;
```

The traversal is type-sensitive: for each statement s, the slicer dispatches control to a dedicated handler H based on the AST node type of s. These handlers are responsible for analysing the structure and semantics of each statement, extracting relevant variable usages, and identifying dependencies.

Each handler contributes to two key aspects of slicing:

1. **Variable Tracking**: Analysing variable occurrences in s, updating the Variable Resolution Tracker $VRT = (R, U)$, depending on whether variables are assigned or merely used.

2. **Dependency Resolution**: Identifying statements s' that define values required by s, and updating the Dependency Graph DG by adding edges (s, s', σ) as discussed in Section 4.2.1.

This logic is formalized in Algorithm 4.3, which illustrates the typical behaviour implemented by a handler.

Algorithm 4.3: Handler Logic for Statement s

Input: Statement $s \in V_{AST}$, Dependency Slicer $S = (DG, VRT = (R, U), \dots)$

Identify the syntactic role of s (e.g., assignment, import, conditional, ...) ;
Infer variables in s and classify them as definitions or usages ;
foreach *defined variable* x *in* s **do**
    Add x to R ;
    Remove x from U if present ;
end
foreach *used variable* y *in* s *such that* $y \notin R$ **do**
    Add y to U (unresolved variables) ;
end
foreach *dependency* $s' \in V_{AST}$ *referenced by* s **do**
    Add edge (s, s', σ) to DG ;
    `visit`(s') ;
end

This recursive resolution mechanism ensures that all direct and transitive dependencies of the RUIs are captured. Consequently, the constructed slice $P_S \subseteq P$ includes all statements necessary for the semantically correct execution of the selected range.

Due to the scope of this work, only a selected set of statement types are handled explicitly, including annotated assignments, function and class definitions, conditionals, expressions, and imports. These cover the most common slicing scenarios. The handler-based architecture is modular and extensible. New handlers can be added by following the general logic formalized in Algorithm 4.3, adapting the behaviour to the semantics of the specific statement type.

In addition to dependency tracking, each handler also performs type inference for all variables encountered during traversal. For each variable added to the variable resolution tracker, whether to the resolved set R or the unresolved set U, a type τ is inferred based on type annotations, function signatures or constructor definitions. This type τ is stored alongside each variable in the VRT, ensuring that every entry in $R \cup U$ carries an associated type. These types are used in later phases to support typed user interaction (see Section 4.3).

In the following subsections, we detail the behaviour of each supported handler. To illustrate how the traversal and handler logic operate across multiple statement types in practice, the final subsections walks through the complete slicing process for both the simple and complex examples introduced in Section 4. These walkthroughs concretely demonstrate how the dependency graph DG, the variable resolution sets R and U, and the recursive invocation of handlers interact to construct the final slice.

Annotated Assignments (`AnnAssign`)

An annotated assignment is a statement of the form:

$$x : \tau = e$$

where x is a variable name, τ is a type annotation, and e is an expression.

The handler $H_{AnnAssign}$ performs the following steps:

1. The left-hand side variable x is treated as a *definition*. The handler adds x to the resolved set R, provided x is assigned within the RUI and not used before its assignment.
2. If x appears in a usage context prior to its assignment (e.g., in a condition or earlier expression), it is instead added to U.
3. The right-hand side expression e is recursively analysed to extract variable usages:
   - For each variable y occurring in e, if $y \notin R$, then y is added to U.
   - If y is an imported symbol, resolve the corresponding `Import` or `ImportFrom` statement s_{imp}, add the edge (s, s_{imp}, y) to DG, and invoke `visit`(s_{imp}).
   - If e involves a function call $f(y_1, \ldots, y_n)$, then:
     - The arguments y_i are marked as unresolved, unless already in R.
     - A dependency is recorded to the function definition of f, i.e., $(s, s_f, \texttt{f}) \in E_{DG}$, where s_f defines f.
     - `visit`(s_f) is invoked to ensure that the referenced function is itself analysed and its dependencies resolved.
   - If e refers to a user-defined class C (e.g., as a type annotation or constructor), resolve the corresponding `ClassDef` or `Import` statement s_C, add the dependency edge (s, s_C, C) to DG, and invoke `visit`(s_C).

This rule ensures that all variables and types referenced in an assignment are tracked, and that dependent statements (functions, classes, imports) are recursively included in the slice.

In the simple example from Listing 4.1, the assignment `message: str = "hello"` on line 1 adds the variable `message` to R. Since the expression `"hello"` is a literal and contains no references to other symbols, no additional dependencies are recorded.

To illustrate a more complex case, consider the annotated assignment `user: User = User(name, is_admin)`. Although not part of the complex example program in Listing 4.3, this example demonstrates the full capabilities of the handler. The variable `user` is added to R and the class `User` is referenced both in the type annotation and the constructor call. Accordingly, a dependency edge $(s, s_{User}, \texttt{User})$ is added to DG,

where s_{User} is the `ClassDef` or the corresponding `Import` statement that introduces `User`. Subsequently, `visit`(s_{User}) is invoked to ensure that the referenced class is itself analysed and its dependencies resolved. Additionally, the constructor arguments `name` and `is_admin` are added to U, unless already present in R.

Expression Statements (`Expr`)

Expression statements (`Expr`) are standalone expressions that are not assigned to any variable. They often include function calls for side effects, such as printing to the console or logging.

When processing a statement s of type `Expr`, the handler H_{Expr} performs the following steps:

1. Analyse the expression e for variable usages and function calls:
   - Each variable y used in e is checked against the resolved set R.
   - If $y \notin R$, it is added to the unresolved set U.
2. If y is an imported symbol, resolve the corresponding `Import` or `ImportFrom` statement s_{imp}, add the edge (s, s_{imp}, y) to DG, and invoke `visit`(s_{imp}).
3. If e is a function call $f(y_1, \ldots, y_n)$:
   - Each argument y_i is added to U unless already present in R.
   - A dependency edge $(s, s_f, \texttt{f})$ is added to DG, where s_f defines f.
   - `visit`(s_f) is invoked to recursively analyse the function definition.
4. If e involves a method call $x.m()$, and x has a custom class type C:
   - Resolve C to the corresponding `ClassDef` or `Import` statement s_C.
   - Add the dependency edge (s, s_C, C) to DG.
   - Invoke `visit`(s_C).

This handler contributes primarily to usage tracking and the discovery of transitive dependencies via function and method calls.

In the simple example from Listing 4.1, the expression `print(message.upper())` on line 2 is an expression statement. Here, `message` is marked as unresolved $U \leftarrow U \cup \{message\}$ since it appears in the RUI without its assignment. The call to `upper()` is a method on a built-in type and does not introduce additional dependencies.

In the more complex example, the expression `log_access(user)` on line 10 in Listing 4.3 is handled as follows: The variable `user` is added to U unless already present in R. A dependency edge $(s, s_{\texttt{log\_access}}, \texttt{log\_access})$ is added to DG, where $s_{\texttt{log\_access}}$ is the definition of the `log_access` function (line 3). Subsequently, `visit`($s_{\texttt{log\_access}}$) is invoked to resolve transitive dependencies of that function.

Import Statements

Import statements introduce external symbols into the local scope. They can take several syntactic forms, such as `import x`, `import x as y`, or `from m import x`. All import statements are normalized, as described in Section 4.2.1, into the canonical form:

$$\texttt{import}\ x\ \texttt{as}\ x \quad \text{or} \quad \texttt{from}\ m\ \texttt{import}\ x\ \texttt{as}\ x$$

This ensures uniform handling of imported symbols throughout the slicing process.

When processing a normalized import statement s, the handler H_{Import} performs the following steps:

1. Determine whether the imported module can be resolved to a local source file.
2. If resolvable, retrieve the corresponding AST and locate the defining statement s_{def} of the imported symbol.
3. Add a dependency edge (s, s_{def}, x) to DG, where x is the imported symbol.
4. Invoke `visit`(s_{def}) to transitively resolve dependencies associated with x.

If the imported module cannot be resolved (e.g., if it belongs to the standard library or lacks accessible source code), the handler does not record a dependency edge or invoke further traversal.

This handler enables the slicer to incorporate external source-level definitions introduced through import statements.

As an example, consider the statement `from user import User` on line 1 in Listing 4.3. This introduces the symbol `User` into the current file. The handler locates the `ClassDef` statement $s_{\texttt{User}}$ in the imported file `user.py` (line 1), adds a dependency edge $(s, s_{\texttt{User}}, \texttt{User})$ to DG, and invokes `visit`$(s_{\texttt{User}})$ to ensure its dependencies are recursively analysed.

Function and Class Definitions (`FunctionDef`, `ClassDef`)

Function and class definitions encapsulate reusable program logic and data structures. They frequently appear as dependency targets in function calls, parameter annotations, inheritance structures, or internal references.

The handler $H_{\texttt{ClassFunc}}$ performs the following steps when processing a statement $s \in V_{AST}$ of type `FunctionDef` or `ClassDef`:

1. For function definitions:
   - For each parameter p_i in the function signature, infer its type annotation τ_i, if present.

- If τ_i refers to a user-defined class or imported symbol, identify the corresponding defining statement $s_{\tau_i} \in V_{AST}$, add a dependency edge $(s, s_{\tau_i}, \tau_i) \in E_{DG}$, and invoke `visit`$(s_{\tau_i})$.

2. For class definitions:
   - For each superclass B, identify the defining statement $s_B \in V_{AST}$, add $(s, s_B, B) \in E_{DG}$, and invoke `visit`(s_B).
3. Analyse the body of the function or class definition:
   - Traverse annotations, assignments, and expressions to identify referenced symbols (e.g., constructors, types, field accesses).
   - For each referenced symbol x, identify its origin $s_x \in V_{AST}$, add a dependency edge $(s, s_x, x) \in E_{DG}$, and invoke `visit`(s_x).
4. Query the call graph G_{CG} to identify all callees $f_1, \ldots, f_n$ that are transitively invoked from within the body:
   - For each callee f_j, resolve its definition $s_{f_j} \in V_{AST}$, add $(s, s_{f_j}, f_j) \in E_{DG}$, and invoke `visit`$(s_{f_j})$.
5. Identify internal import statements located within the body (e.g., `import` inside a function) and process them by invoking `visit` on their corresponding AST nodes s_{imp}.

This handler ensures that definitions, referenced types, and transitive dependencies introduced via function calls, class hierarchies, or annotations are fully tracked and integrated into the dependency graph DG.

In the complex example in Listing 4.3, the function `process_user` defined on line 6 is handled as follows:

- Its parameter `user: User` triggers resolution of the class `User`, leading to a dependency edge $(s, s_{\texttt{User}}, \texttt{User}) \in E_{DG}$, followed by a recursive call `visit`$(s_{\texttt{User}})$.
- The function body contains a call to `log_access(user)`, triggering resolution of `log_access` to its definition on line 3, and adding the new edge $(s, s_{\texttt{log\_access}}, \texttt{log\_access}) \in E_{DG}$ to DG, followed by `visit`$(s_{\texttt{log\_access}})$.

Conditional Statements (`If`)

Conditional statements influence control flow and introduce branching logic based on runtime conditions. They require careful treatment in slicing to ensure that all relevant branches and condition expressions are properly analysed.

A generic `if`-statement has the following form:

```
if e:
    body_if
else:
    body_else
```

Here, e is the condition expression, and the two blocks represent the bodies of the `if` and `else` branches, respectively. The else part may be omitted, or consist of a nested `if` (`elif`).

The handler $H_{\texttt{If}}$ processes a statement $s \in V_{AST}$ of type `If` as follows:

1. Instantiate three local variable resolution trackers, each represented as a tuple $T = (R_T, U_T)$:
   - $T_{\texttt{cond}} = (\{\}, \{\})$: tracks resolution in the condition expression e,
   - $T_{\texttt{if}} = (\{\}, \{\})$: tracks resolution in the `if` branch,
   - $T_{\texttt{else}} = (\{\}, \{\})$: tracks resolution in the `else` branch (if present).

   These local trackers allow the handler to reason independently about resolution behaviour in each branch. Variables are only added to the global resolved set R if they are consistently resolved across all branches (i.e., present in both $R_{T_{\texttt{if}}}$ and $R_{T_{\texttt{else}}}$ if applicable). All other variables are added to U.
2. Analyse the condition expression e:
   - For each variable y used in e, if $y \notin R_{T_{\texttt{cond}}}$, then add y to $U_{T_{\texttt{cond}}}$.
   - If e involves a function or method call (e.g., $f(y_1, \ldots, y_n)$ or $x.m()$):
     - For each argument y_i, add y_i to $U_{T_{\texttt{cond}}}$ unless $y_i \in R_{T_{\texttt{cond}}}$.
     - Locate the definition $s_f \in V_{AST}$ of the function or method and add the edge $(s, s_f, f) \in E_{DG}$.
     - Invoke `visit`(s_f).
   - If e references a class-defined attribute or method (e.g., $x.m()$) and x has type C, resolve the class C, locate its definition $s_C \in V_{AST}$, add $(s, s_C, C) \in E_{DG}$, and invoke `visit`(s_C).
3. Analyse each body of the conditional:
   - For each statement in the `if` body, invoke `visit` passing $T_{\texttt{if}}$.
   - For each statement in the `else` body (if present), invoke `visit` passing $T_{\texttt{else}}$.

   The traversal distinguishes between fully and partially included conditionals:

- If the entire `if`-statement lies within the RUI, its structure is preserved in the final slice, and its child statements are treated as part of a grouped control structure.
- If only parts of the body are within the RUI, the `if` structure itself is omitted, and only the relevant sub-statements are included in the slice.

4. Merge the local resolution trackers into the global resolution state:
   - A variable is added to the global resolved set R only if it appears in both $R_{T_{\texttt{if}}}$ and $R_{T_{\texttt{else}}}$.
   - All other variables recorded in any unresolved set $U_{T_{\texttt{cond}}}$, $U_{T_{\texttt{if}}}$, or $U_{T_{\texttt{else}}}$ are added to the global unresolved set U.

This handler ensures that variable resolution remains sound across different control-flow paths by applying branch-local resolution logic and integrating dependencies introduced by variables, function calls, and type annotations in both condition and body statements.

In the complex example from Listing 4.3, the conditional statement on line 7 isanalysedd as follows:

- The condition `user.has_admin_rights()` involves a method call. The variable `user` is added to the unresolved set $U_{T_{\texttt{cond}}}$. Based on its type annotation, the class `User` is identified, leading to a dependency edge $(s, s_{\texttt{User}}, \texttt{User}) \in E_{DG}$, followed by $\texttt{visit}(s_{\texttt{User}})$.
- The method `has_admin_rights` is part of `User`, so its resolution is handled transitively via the visit to the class definition.
- In the `if` branch, the statement `print("Access granted")` is visited, but it introduces no additional dependencies or variable references.
- In the `else` branch, the expression `log_access(user)` is encountered. The variable `user` is added to $U_{T_{\texttt{else}}}$, and a dependency edge $(s, s_{\texttt{log\_access}}, \texttt{log\_access}) \in E_{DG}$ is introduced. The function definition $s_{\texttt{log\_access}}$ is then visited.
- After all components of the conditional are processed, the unresolved sets $U_{T_{\texttt{cond}}}$, $U_{T_{\texttt{if}}}$ and $U_{T_{\texttt{else}}}$ are merged into the global unresolved set U. The resulting global unresolved set is therefore $U = \{user\}$. This ensures that all unresolved variables are retained for later resolution via user input.

Slicing Walkthrough: Simple Example

The simple example, introduced in Section 4 and shown in Listing 4.4, contains the following two statements:

```
1 message: str = "hello"
2 print(message.upper())
```

Listing 4.4: Simple Example Program

The Range Under Investigation consists only of the call to `print` on line 2.

Initial State
Before traversal begins, the slicer is initialized as follows:

- Resolved variables: $R = \{\}$
- Unresolved variables: $U = \{\}$
- Dependency graph: $DG = (\{\}, \{\})$

Slicing Traversal
The slicer begins by applying the handler $H_{\texttt{Expr}}$ to this expression statement. The expression `print(message.upper())` is analysed, and the variable `message` is identified in a usage context. Since there is no prior assignment to `message` within the RUI and it is not already in the resolved set R, it is added to the unresolved set:

$$U \leftarrow \{\texttt{message}\}$$

The method call `message.upper()` is a method on a built-in type (`str`) and does not trigger any dependency resolution.

Final State
The final state looks like:

- Resolved variables: $R = \{\}$
- Unresolved variables: $U = \{\texttt{message}\}$
- Statements in the slice: $P_S = \{s_2\}$
- Dependency graph: $DG = (\{\}, \{\})$

The unresolved variable `message` will later be classified as a *user variable* in the transformation phase described in Section 4.3. Its concrete value will be provided externally.

Slicing Walkthrough: Complex Example

The complex example, introduced in Section 4, spans two modules:

```
1  class User:
2      def __init__(self, name: str, is_admin: bool):
3          self.name = name
4          self.is_admin: bool = is_admin
5
6      def has_admin_rights(self) -> bool:
7          return self.is_admin
8
9      def __str__(self) -> str:
10         return self.name
```

Listing 4.5: File: `user.py`

```
1  from user import User
2
3  def log_access(user: User) -> None:
4      print(f"Access denied for: {user}")
5
6  def process_user(user: User) -> None:
7      if user.has_admin_rights():
8          print("Access granted")
9      else:
10         log_access(user)
11     print("processed user")
12
13 def log_access_attempt() -> None:
14     print("User attempted access")
```

Listing 4.6: File: `access_control.py`

The RUI includes lines 7 to 10 in `process_user`.

Initial State

Before traversal begins, the internal state is initialized as follows:

- Resolved variables: $R = \{\}$
- Unresolved variables: $U = \{\}$
- Dependency graph: $DG = (\{\}, \{\})$

Slicing Traversal

The slicer applies the handler $H_{\texttt{If}}$ to the conditional statement s on line 7, `if user.has_admin_rights()`. The condition involves a method call and the variable `user` appears in a usage context. Since it is neither defined within the RUI nor previously resolved, it is added to the unresolved set:

$$U_{T_{\text{cond}}} \leftarrow \{\texttt{user}\}$$

From the parameter annotation in `process_user`, the class `User` is inferred as the expected type for `user`. The corresponding import statement on line 1 is identified as the defining statement $s_{\texttt{User}}$, resulting in the edge:

$$(s, s_{\texttt{User}}, \texttt{User}) \in E_{DG}$$

This import is then visited, invoking $\texttt{visit}(s_{\texttt{User}})$. Within the import handler, the definition of `User` is resolved to its class statement $s_{\texttt{class}}$ and another dependency is added:

$$(s_{\texttt{User}}, s_{\texttt{class}}, \texttt{User}) \in E_{DG}$$

Finally, $\texttt{visit}(s_{\texttt{class}})$ is invoked. Since the entire class definition is included, the method `has_admin_rights` is transitively available and no additional edge is required at this point.

The slicer continues with the `else` branch on line 10, which contains a call to `log_access(user)`. The variable `user` appears again in a usage context and is added to the branch-local unresolved set:

$$U_{T_{\texttt{else}}} \leftarrow \{\texttt{user}\}$$

The function `log_access` is resolved to its definition on line 3 and the edge

$$(s, s_{\texttt{log\_access}}, \texttt{log\_access}) \in E_{DG}$$

is added. The corresponding function definition is then visited.

After both branches and the condition have been analysed, the local variable resolution trackers

$$T_{\texttt{cond}}, \quad T_{\texttt{if}}, \quad T_{\texttt{else}}$$

are merged. No variable was defined in all branches, so the resolved set remains empty:

$$R \leftarrow \{\}$$

The variable `user` was referenced in both the condition and the else-branch, and is therefore added to the global unresolved set:

$$U \leftarrow U_{T_{\texttt{cond}}} \cup U_{T_{\texttt{if}}} \cup U_{T_{\texttt{else}}} = \{\texttt{user}\}$$

Final State

- Resolved variables: $R = \{\}$
- Unresolved variables: $U = \{\texttt{user}\}$
- Statements in the slice:

$$P_S = \{s_{\texttt{if}}, s_{\texttt{log\_access}}, s_{\texttt{User}}, s_{\texttt{class}}\}$$

- Dependency graph edges:

$$(s, s_{\texttt{log\_access}}, \texttt{log\_access})$$
$$(s, s_{\texttt{User}}, \texttt{User})$$
$$(s_{\texttt{User}}, s_{\texttt{class}}, \texttt{User})$$

The unresolved variable `user` will later be classified as a *user variable* and provided externally, as explained in Section 4.3. Figure 4.2 illustrates the dependency graph after the slicing process.

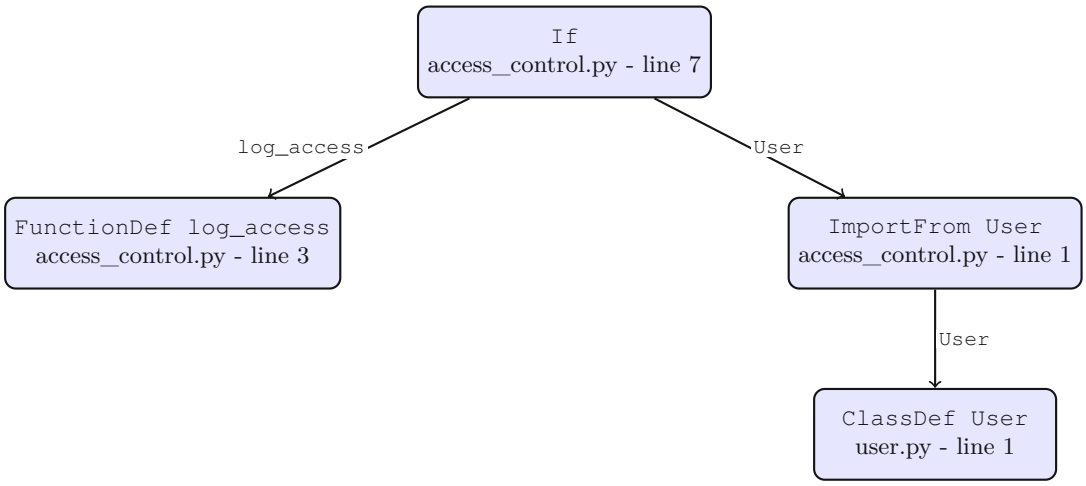

Figure 4.2: Dependency graph constructed during slicing of the complex example.

4.3 User Interaction

After slicing completes, all variables in the unresolved set U are considered *user variables*, meaning they are referenced within the RUI but have no definition in the selected region. Instead of requiring these values to be hardcoded or manually passed in by the user, the tool automatically prompts for input values during execution. This behaviour enables focused execution by allowing users to provide only the minimal set of inputs required for the selected computation. The ability to run the sliced program multiple times with varying inputs offers valuable insight into isolated behaviours of the original code, without the need to construct full test harnesses or invoke complex debugging sessions.

To handle this interaction, a dedicated input mechanism is utilized to request and validate the values of variables in U. For each variable $v \in U$, an inferred type τ is determined during slicing using available type annotations, constructor signatures or structural inference. The input mechanism then generates a tailored prompt that enforces type correctness by validating user responses against τ before incorporating the result into the runtime execution context of the slice.

Each variable $v \in U$ is processed according to its associated type τ, and the interaction logic is adapted accordingly. For primitive types such as `int`, `str`, `float`, and `bool`, the user is prompted to enter a single value which is validated to ensure compatibility with τ. For structured types including `List`, `Set`, `Dict`, and `Tuple`, the user is first asked to specify the number of elements and then prompted for each individual value recursively, with validation based on the component types. If τ is optional or a union of alternatives, the user is asked whether to supply a value and, if applicable, which of the permissible types to use. In cases where τ is a literal type, such as `Literal["A", "B"]`, the user selects a value from the predefined constant set. When τ refers to a user-defined class, values for the required constructor arguments are collected interactively, with prompts generated recursively for each parameter based on its type. The collected data is stored in a structured form that can later be used to instantiate the class.

To illustrate the behaviour of this mechanism, consider the examples introduced in Section 4.

In the simple example, the region under investigation includes a single statement:

```
1 print(message.upper())
```

Since the variable `message` has no definition in the sliced region and is not resolved transitively, it is treated as a user variable with type `str`. The input handler prompts the user to enter a string and validates the provided value. Suppose the user enters `"hello world"`. This value is recorded and later injected into the generated code as shown in Listing 4.7.

In the complex example, the unresolved variable is `user`, whose type is inferred as `User` based on its usage and annotations. The corresponding constructor is:

```
1 def __init__(self, name: str, is_admin: bool)
```

The input handler recursively collects the values for both fields, `name` and `is_admin`. Assume the user provides `"Mallory"` for `name` and `False` for `is_admin`. These inputs are structured into a class instantiation and later used to construct the variable `user` within the `main` block of the generated code (Listing 4.8).

This user input phase guarantees that all unresolved variables are resolved to concrete runtime values in a type-safe manner. The validation mechanism applies uniformly across primitive values, nested data structures, and class instances, ensuring that all inputs conform to the semantics and expectations of the original program.

4.4 Code Generation

Once unresolved variables have been resolved through user input (Section 4.3), the extracted slice $P_S \subseteq P$, obtained through dependency analysis, is now transformed into a runnable program. This transformation phase converts the dependency-resolved slice into a concrete collection of source files, suitable for direct execution.

To ensure semantic correctness and runtime completeness, the code generation procedure performs the following steps. First, it computes a topological order over the statements in the slice using the dependency graph $DG = (V_{DG}, E_{DG})$, ensuring that each usage is preceded by its corresponding definition. Second, it partitions these statements by file and serializes them to source code in dependency-respecting order. Third, it emits the output to new files, injecting user-provided values and wrapping the region of interest in a callable function. Control flow constructs like `if`-statements are reconstructed when only partial sub-statements are selected. Finally, all imports are updated to point to the rewritten files, preserving cross-file linkage.

This process is shown in Algorithm 4.4 and detailed below.

Algorithm 4.4: Slice Extraction and Code Generation

Input: Dependency Graph $DG = (V_{DG}, E_{DG})$, Resolved User Variables U
Output: Mapping $\mathcal{C} : F \to \texttt{str}$

Let $P_S \leftarrow V_{DG} \cup (RUI \setminus V_{DG})$;
Partition $P_S = \bigcup_{f \in F} V_f$, compute $\prec_{DG}$ and $\prec_{\text{files}}$;
Initialize $\mathcal{C}(f) \leftarrow$ `""` $\quad \forall f \in F$;
foreach $f \in F$ *in order* $\prec_{files}$ **do**
    /* Global User Variable Injection */
    foreach $v \in U$ *with* $Scope(v) = global \wedge File(v) = f$ **do**
        Append `v.name = v.repr_value()` to $\mathcal{C}(f)$;
    end
    /* Statement Serialization and Conditional Reconstruction */
    foreach $s \in V_f$ *in order* $\prec_{DG}$ **do**
        if s *is partial* `if`*-statement* **then**
            Reconstruct s' with only selected branches ;
        end
        Append `code`(s) or `code`(s') to $\mathcal{C}(f)$;
    end
    ;
    /* Wrapping the RUI in a Callable Function */
    if f *is the main file* **then**
        Wrap RUI in `wrapped_code` with $v_1, \ldots, v_k \in U$ as parameters ;
        Append a `main` block with $v_i =$ `repr_value`(v_i) and invocation ;
    end
    /* Import Rewriting */
    Update all import statements in $\mathcal{C}(f)$ to match rewritten file paths ;
end
return $\mathcal{C}$

Inputs and Preliminaries

The input to the code generation phase consists of:

- The dependency graph $DG = (V_{DG}, E_{DG})$, where $V_{DG} \subseteq V_{AST}$ contains all visited statements and $E_{DG} \subseteq V_{DG} \times V_{DG} \times \Sigma$ captures the dependency edges.
- The unresolved variable set U, now resolved with concrete values through user interaction.
- The user-selected range under investigation, $RUI \subseteq V_{AST}$, which may include nodes not in V_{DG}.

The full program slice to be emitted is then defined as:

$$P_S = V_{DG} \cup (RUI \setminus V_{DG})$$

Statement Ordering

Correct execution requires that all statements in P_S are ordered such that definitions precede their use. This is ensured by a topological sort over the dependency graph DG. The induced partial order $\prec_{DG}$ is defined as:

$$(s_i, s_j, \sigma) \in E_{DG} \Rightarrow s_i \prec_{DG} s_j$$

This ensures that if statement s_i depends on statement s_j, then s_j appears before s_i in the emitted output.

For example, the simple example introduced in Section 4 contains no dependency edges, and the set V_{DG} remains empty. The only required element is the unresolved variable `message`, which is handled separately and does not influence the ordering of statements. Hence, no topological sort is required.

In contrast, the complex example from Section 4 yields a non-empty dependency graph. A topological sort yields the order shown below, where each statement must appear after its dependencies:

```
if user.has_admin_rights() (access_control.py, line 7)
 ≺DG log_access (access_control.py, line 3)
 ≺DG import User (access_control.py, line 1)
 ≺DG class User (user.py, line 1)
```

File Partitioning and Output Mapping

The sorted statements are grouped by source file. Let F be the set of files referenced in P_S. For each $f \in F$, define:

$$V_f = \{s \in P_S \mid \text{File}(s) = f\}$$

A topological order is also computed over the file set to respect inter-file dependencies:

$$f_i \prec_{\text{files}} f_j \quad \text{if} \quad \exists (s_i, s_j, \sigma) \in E_{DG} \text{ such that } \text{File}(s_i) = f_i,\ \text{File}(s_j) = f_j$$

The output is represented as a mapping $\mathcal{C} : F \to \texttt{str}$, where $\mathcal{C}(f)$ accumulates the generated source code for file f.

For instance, in the simple example, only a single file is involved, so no cross-file sorting is necessary: $F = \{\texttt{simple.py}\}$.

In the complex example, the class `User` from `user.py` is required by `access_control.py`, due to the import in line 1. This results in the topological file order `access_control.py` $\prec_{\text{files}}$ `user.py`, ensuring correct placement of statements during reconstruction.

Global User Variable Injection

All variables $v \in U$ with global scope are declared at the top of their respective files. Let $\texttt{Scope}(v) = \text{global}$ and $\text{File}(v) = f$. Then:

$$\mathcal{C}(f) \leftarrow \mathcal{C}(f) \,\|\, \texttt{v.name} = \texttt{v.repr\_value()}$$

Here, `v.name` denotes the identifier of the variable and `v.repr_value()` produces its serialized runtime value in the specific programming language syntax.

For example, in the simple example, the variable `message` is global and initialized as `message = "hello world"` at the top of the generated file (Listing 4.7, line 1).

In contrast, the complex example involves only local user variables, so no declarations are inserted during this phase.

Statement Serialization and Conditional Reconstruction

Statements in V_f are emitted in topological order. If $s \in V_f$ is a compound `if`-statement such that only a subset of its body or orelse branch is included in P_S, it is reconstructed to retain valid control flow:

$$\texttt{if } e\texttt{:} \quad \{s_i \in \texttt{body}(s) \cap P_S\} \quad \texttt{else:} \quad \{s_i \in \texttt{orelse}(s) \cap P_S\}$$

where s_i denotes any sub-statement of the original conditional block. If the full structure is included, the original form is preserved.

In the simple example, the only relevant statement is a single print call, so this step requires no reconstruction.

By comparison, the complex example includes an if-statement spanning lines 7 to 10. Since the entire body and else-branch are included in P_S, the conditional is retained in its original form and inserted directly into the wrapped function.

Wrapping the RUI in a Callable Function

In the main file (i.e., the file that contains the RUI), all local user variables are passed as parameters to a generated function named `wrapped_code`. The RUI is embedded as the function body, and the values for the user variables are assigned in a standard `main` block.

Let $v_1, \ldots, v_k \in U$ be the local user variables in the main file f_{main}. The generated function is:

$$\texttt{def wrapped\_code}(v_1, \ldots, v_k)\texttt{:}$$
$$\texttt{<RUI statements>}$$

At the end of the file, the following entrypoint is appended:

$$\texttt{main():}$$
$$\texttt{v}_i = \texttt{repr\_value}(v_i) \quad \forall i \in [1, k]$$
$$\texttt{wrapped\_code}(v_1, \ldots, v_k)$$

The overall pattern is illustrated by the following sketch:

```
1 def wrapped_code(...):
2     <RUI statements>
3
4 if __name__ == '__main__':
5     <user variable initializations>
6     wrapped_code(...)
```

For example, the simple example generates the function `wrapped_code()` without parameters (line 3) and a main block that directly invokes it (line 7).

For the complex example, the local variable `user` is passed as a parameter to `wrapped_code` in line 6 and initialized in the main block on line 13. The call is then issued in line 14, completing the wrapped execution setup.

Import Rewriting and Final Emission

All import statements in the generated files are rewritten to match the new paths of the rewritten modules. This ensures that cross-file references remain valid and executable. Let $\phi : \text{OriginalPath} \to \text{NewPath}$ be a mapping from original module locations to rewritten files. Each import line is updated as:

$$\texttt{import } m \quad \mapsto \quad \texttt{import } \phi(m)$$

This guarantees that inter-file references remain valid and that the resulting program is self-contained and executable.

For example, in the complex example, line 1 of the output file updates the import to `from new_user import User`, reflecting the location of the newly generated file.

Bringing all steps together, the resulting output for both the simple and the complex example, illustrates how the reconstructed program files reflect the semantics of the sliced region while eliminating unrelated code.

In the simple example, the resulting file, as shown in Listing 4.7, contains only the print statement explicitly selected by the user, wrapped in a function and invoked from a `main` block. The unresolved variable `message` is treated as a global user variable and declared at the top of the file. No additional dependencies are introduced, and the slice remains minimal.

In contrast, the complex example features multiple functions and class definitions, from which only a subset is required for executing the selected code region. Listings 4.8 and 4.9 show the final emitted files. The class `User`, originally declared in `user.py` (line 1), is preserved in the output `new_user.py` to satisfy cross-file dependencies. Meanwhile, the main logic from `access_control.py` is isolated into a `wrapped_code` function (line 6) and invoked from the main block (line 14) after initializing the required user input (line 13). The slice correctly omits the call to `log_access_attempt` and the final print statement from the original program (lines 13 and 11), which are unrelated to the selected computation.

These listings illustrate that the slicing transformation correctly preserves program behaviour for the selected region, capturing only relevant dependencies and user inputs while omitting unrelated code.

```
1  message = "hello world"
2
3  def wrapped_code():
4      print(message.upper())
5
6  if __name__ == '__main__':
7      wrapped_code()
```

Listing 4.7: File `new_simple_example.py` – Generated Code for the Simple Example

```
1  from new_user import User
2
3  def log_access(user: User) -> None:
4      print(f"Access denied for: {user}")
5
6  def wrapped_code(user):
7      if user.has_admin_rights():
8          print("Access granted")
9      else:
10         log_access(user)
11
```

```
12 if __name__ == '__main__':
13     user = User(name="Mallory", is_admin=False)
14     wrapped_code(user)
```

Listing 4.8: File `new_access_control.py` – Generated Code for the Complex Example (Main File)

```
1  class User:
2      def __init__(self, name: str, is_admin: bool):
3          self.name = name
4          self.is_admin: bool = is_admin
5
6      def has_admin_rights(self) -> bool:
7          return self.is_admin
8
9      def __str__(self) -> str:
10         return self.name
```

Listing 4.9: File `new_user.py` – Generated Code for the Complex Example (Dependency File)

4.5 Execution of the Generated Code

Following code generation (Section 4.4), the resulting slice is executed as a standalone program. This step concludes the slicing workflow by executing the reconstructed code using the provided user inputs and resolved dependencies.

Execution is initiated from the main file f_{main}, as defined in Section 4.4, which contains the RUI. This file includes a `main` block that binds all unresolved variables U to their user-provided values and calls the wrapped function containing the RUI.

The overall behaviour of the program is determined by the region under investigation, the resolved variables U, and the dependency structure encoded in the graph DG. This step provides the final output of the slicing process in the form of executable behaviour.

The execution outcome depends directly on the inputs provided by the user and the logic included in the selected slice.

The behaviour of this phase is illustrated using both the simple and complex example introduced earlier.

In the simple example (Section 4), the file `new_simple_example.py` contains the global variable `message`, which was resolved by the user to `"hello world"`. Upon execution, the wrapped function is invoked, evaluating the single print statement:

```
1 print(message.upper())
```

This produces the output:

```
1 HELLO WORLD
```

In the complex example (Section 4), the main file is `new_access_control.py`. The user provided the values `name = "Mallory"` and `is_admin = False` for the unresolved variable `user`, which is instantiated in the `main` block. The call to `wrapped_code(user)` triggers a conditional check:

```
if user.has_admin_rights():
    print("Access granted")
else:
    log_access(user)
```

Since the user input yields `False` for `is_admin`, the else branch is taken and the function `log_access` is called, resulting in the printed output:

```
Access denied for: Mallory
```

The execution finalizes the workflow by producing a runnable program that accurately reflects the intent and scope of the selected region.

CHAPTER 5

CodeDetective

This chapter presents the concrete implementation of the slicing approach introduced in Chapter 4, focusing on those aspects that require special consideration in practice. Rather than restating the entire workflow, this chapter highlights the key implementation decisions, data structures, and supporting infrastructure that make the approach executable. CodeDetective is implemented as a command-line tool written in Python, designed for typed Python projects. It integrates external analysis libraries and builds on open-source infrastructure to support static slicing, user interaction, and isolated execution of the selected code region.

5.1 Foundation

CodeDetective builds upon the existing open-source framework Repository to Environment (R2E) [19], which converts GitHub repositories into interactive execution environments for testing and analysing code generation systems and programming agents at scale. It extracts functions and methods from the repository, using Large Language Models (LLM)s to create and run equivalence tests, and setting up an environment where the generated code can be executed and evaluated. Our tool leverages its `pat` module, which provides foundational program analysis capabilities, including Abstract Syntax Tree (AST) processing and dependency analysis. We extend and adjust this framework to support advanced program dependency slicing tailored to our use case. The original `r2e` project is publicly available at `https://github.com/r2e-project/r2e`.

5.2 Command-Line Interface and Setup

The tool is invoked via a command-line interface. The user specifies three arguments:

- `-program_path`: Path to the root directory of the project

- `-file_under_investigation_path`: Path to the file that contains the RUI
- `-line_number_under_investigation`: One or more line numbers marking the RUI

An example call is shown below:

```
python codedetective.py
  --program_path /home/user/project/
  --file_under_investigation_path /home/user/project/example.py
  --line_number_under_investigation 107-112
```

5.3 Repository Initialization

Upon execution, the tool clones the input project to a temporary working directory and adjusts all internal paths accordingly. This guarantees that any analysis or transformation is isolated from the original source.

To support interprocedural slicing, the tool constructs a static call graph of the entire project. This is implemented using the `pycg` library [26], which analyses Python code statically and generates a graph $G_{CG} = (V_{CG}, E_{CG})$ where vertices correspond to functions and methods and edges indicate possible call relationships. This call graph is a crucial component for dependency analysis, as described in Section 4.2.

In addition to the call graph, a class registry is created during the setup phase. This registry is a global mapping that indexes all user-defined classes by their fully qualified name, consisting of module path and class name. Each entry maps to the AST node representing the class and its source file.

The class registry plays a crucial role in resolving class-based dependencies during slicing. For example, consider the complex case from Section 4, where the `access_control.py` file contains the line 6:

```
1 def process_user(user: User) -> None:
```

Here, `User` refers to a class defined externally in `user.py`. During slicing, this reference is resolved by consulting the class registry, which includes an entry of the form:

```
user.User → (ClassDef node, user.py)
```

This mapping enables the system to locate the class definition, analyse its structure if needed, and ensure the appropriate import is reconstructed in the generated slice.

The registry is also used during runtime input collection (Section 4.3), where user-defined classes may need to be instantiated. Constructor parameters are extracted from the `__init__` method of the class via its AST node, allowing recursive input prompts to be generated.

5.4 Variable Tracking and Resolution

To support dependency analysis and user input validation, the system maintains a central data structure known as the Variable Usage Tracker (VUT).

The VUT collects all variable identifiers occurring in the program, along with their scope, type, and usage sites.

The tracker is initialized during a preprocessing phase, where the entire Abstract Syntax Tree of the program is scanned to extract variables referenced in each statement. Each variable is recorded in the VUT and associated with:

- its syntactic scope (e.g., module, function, or class body)
- its usage sites $\{s_1, s_2, \ldots\} \subseteq P$
- its data type τ, inferred from annotations, constructor arguments or context

The resulting structure maps each variable v to a tuple:

$$v \mapsto (\tau, \text{scope}, \{s_1, s_2, \ldots\})$$

For example, in the simple example from Section 4, the VUT contains an entry for `message`:

$$\texttt{message@global} \mapsto (\texttt{str}, \texttt{global}, \{\text{line } 1, \text{line } 2\})$$

This indicates that `message` is a global variable of type `str` used on both lines.

In the complex example from Section 4, the variable `user` appears in two distinct scopes. Accordingly, the VUT contains two separate entries:

$$\texttt{user@log\_access} \mapsto (\texttt{User}, \texttt{log\_access}, \{\text{line } 3\})$$

$$\texttt{user@process\_user} \mapsto (\texttt{User}, \texttt{process\_user}, \{\text{line } 6, \text{line } 7, \text{line } 10\})$$

This structure is consulted throughout the slicing process to support the variable tracking step within each statement handler. Handlers query the VUT to identify all variables referenced in a given statement and determine whether they are already resolved. Unresolved variables are added to the unresolved set U and their associated types τ guide the input mechanism in generating appropriate prompts and validating the provided values by the user.

These implementation details demonstrate how the abstract slicing model has been translated into a working system. By grounding the theoretical phases of the approach in concrete mechanisms, the resulting tool enables practical, dependency-aware execution of selected code regions across Python projects.

CHAPTER 6

Evaluation Methodology

This chapter describes the methodology used to evaluate CodeDetective. The evaluation is designed to assess three key aspects: the accuracy of CodeDetective in executing code in isolation, its performance compared to an existing tool and its conceptual differences from similar approaches. To achieve this, we follow a structured evaluation process that includes an iterative testing approach, comparative analysis and a conceptual discussion of CodeDetective compared to similar existing tools.

The goal of the evaluation section is to answer the following evaluation questions:

1. EQ_1**: How accurate is CodeDetective at executing code in isolation?** The evaluation compares execution results with expected outputs to measure accuracy.
2. EQ_2**: How does CodeDetective perform compared to an existing tool (Code Runner [20]) in terms of accuracy and success rate?** The same test cases are executed with both tools and their results are compared.
3. EQ_3**: How does CodeDetective conceptually differ from other similar tools?** The evaluation highlights differences in concepts of CodeDetective compared to Code Runner and LExecutor [27].

6.1 Evaluation Process

To evaluate EQ_1 and EQ_2, we employ an iterative evaluation process in which test cases are structured into levels of increasing complexity. Each test case is derived from a synthetic benchmark project specifically created for this evaluation. Within this project, a specific code range, the RUI, is selected for each test case. This range identifies the statements under investigation and is carefully chosen to match the characteristics of a specific difficulty level.

The evaluation process consists of two main components:

- **Measuring Execution Accuracy (EQ_1)**: We measure the correctness rate of CodeDetective by comparing its output to the expected result. If the actual output matches the expected output, it is considered correct.
- **Comparing Performance Against Code Runner (EQ_2)**: We run the same test cases through both CodeDetective and Code Runner. We compare their success rates (whether the execution completes without errors) and correctness rates (whether the output is correct).

By progressively increasing the complexity of test cases, we identify the limitations of CodeDetective and determine how it performs relative to an established execution tool.

6.2 Difficulty Levels

The test cases used in this evaluation are categorized into ten conceptual levels, each designed to target a specific slicing challenge. The levels progress from simple standalone statements to complex scenarios involving interprocedural calls, external effects and user input. Each new level introduces an additional execution challenge or feature, while still incorporating the capabilities required in previous levels. This cumulative structure ensures that CodeDetective is not only tested in isolated scenarios but also evaluated on its ability to handle multiple interacting features.

Each of the first nine levels includes exactly five test cases, while the tenth level contains seven due to the increased complexity at that stage.

The remainder of this section describes each level in detail, outlining its role within the evaluation and the specific slicing challenges it introduces. Concrete test cases representative of each level are discussed in Section 7.1 and the complete test cases and code snippets listing can be found in the appendix A.

Level 1 – Basic Code Execution (Single & Multi-Line, No Dependencies)

Level 1 targets the most elementary cases of code slicing: isolated statements that are syntactically and semantically self-contained. These include single-line print calls or basic expressions involving literals and constants. The goal at this level is to verify whether the slicer correctly preserves such statements in their entirety, without attempting to infer or introduce extraneous dependencies. Although this level appears trivial, it serves to confirm that the basic parsing and output generation mechanisms function correctly in the absence of variable resolution or control structures.

Level 2 – Variable Dependencies and Simple Expressions

At Level 2, CodeDetective is evaluated in terms of its ability to resolve dependencies between statements that exist within the selected region. This includes code where variable

assignments precede expressions that rely on their values. The central requirement is that the slicer preserve execution order and retain any prior statements that contribute to the definition of later variables. This level confirms that basic data dependencies are detected and resolved locally.

Level 3 – Handling Conditional Statements

Level 3 introduces conditional logic through constructs such as `if`, `elif`, and `else` blocks, possibly in nested or chained form. CodeDetective must not only identify which parts of the conditional structure are relevant to the region of interest but also include any conditions that determine their execution. This includes variables used in the condition expressions, which must be resolved even if they are not themselves part of the output. The level emphasizes control-flow awareness, correct branch inclusion, and path-sensitive dependency handling.

Level 4 – Functions (Simple and Complex)

Level 4 focuses on code that invokes locally or externally defined functions. These functions may require arguments, return computed results, or include nested logic. CodeDetective must recognize which functions are relevant based on the selected region and include their full definitions accordingly. This level tests whether interprocedural relationships are handled correctly and whether function calls are supported within sliced output.

Level 5 – Modules and Imports

Code in Level 5 depends on symbols introduced via import statements. These may originate from the standard library, third-party packages, or project-local modules. The slicing process must determine which imported entities are actually referenced in the selected region and retain the corresponding import statements. The challenge lies in identifying only the necessary imports while omitting unused ones, and in handling different styles of import declarations such as direct, aliased, or from-imports. Additionally, CodeDetective must track any dependencies introduced through the use of imported symbols.

Level 6 – Object-Oriented Code

Level 6 introduces object-oriented constructs, including class instantiation, attribute access, and method calls. CodeDetective needs to ensure that all relevant class declarations and their methods are retained to enable correct behaviour. This level assesses whether the system can handle object lifecycles and include the structural components necessary for execution, especially in the presence of inheritance or encapsulation.

Level 7 – User Input (Primitive Types)

In Level 7, slicing operates on code that references unresolved primitive variables such as integers, strings, booleans, or floats. These values are not defined within the selected region but are required for correct execution. CodeDetective must therefore prompt the user to provide inputs of the appropriate type, enforcing that the provided values match the expected types. The goal is to assess whether undefined values can be detected automatically and whether type validated input collection is supported during slicing.

Level 8 – User Input (Complex Types)

Level 8 extends the user input model to structured types, including collections such as lists and dictionaries, as well as user-defined class instances. These more complex values often require hierarchical input (e.g., populating multiple attributes or nested structures). The slicing logic must identify the shape of the missing values and generate appropriate input prompts or construction logic. This level evaluates whether complex and user-defined types can be meaningfully reconstructed from unresolved values.

Level 9 – API Interactions

Level 9 introduces external communication through API requests. Test cases in this category include constructing and sending HTTP requests and accessing the returned response. CodeDetective needs to include the request logic and retain all statements necessary to make and interpret the API call. This level tests whether external effects such as network interaction are correctly preserved.

Level 10 – File and Database Access

Level 10 includes code that performs persistent I/O operations, such as file access or database modification. These operations may rely on user-supplied paths or connection parameters. The slicing process must retain all relevant file or database interaction logic, including opened resources and manipulated content. This level combines the challenges of external effect handling, input resolution, and program structure preservation under I/O constraints.

6.3 Metrics

For EQ_1, accuracy is evaluated using the correctness rate CR, which measures how often the output of CodeDetective matches the expected result. The correctness rate CR is defined as:

$$CR = \frac{\text{Number of correctly executed test cases}}{\text{Total test cases}} \times 100$$

For EQ_2, we compare the success rate SR and correctness rate CR of CodeDetective against Code Runner. The success rate SR measures how many test case execute without

errors. It is defined as:

$$SR = \frac{\text{Number of successfully executed test cases}}{\text{Total test cases}} \times 100$$

6.4 Test Case Selection and Generation

Initially, the plan was to derive test cases from well-known open-source Python projects available on GitHub[1], including `Black`[2], `Flask`[3], and `Scrapy`[4]. These repositories were also used in the evaluation of LExecutor [27], providing a diverse set of real-world codebases.

However, during the preparation of the evaluation, it became clear that real-world projects are not suitable as a foundation for a meaningful and fair evaluation of CodeDetective. The current implementation of CodeDetective supports a defined and limited subset of Python constructs, namely annotated assignment statements, import statements, class and function definitions, expression statements, and `if`-statements. In real-world codebases, however, it is extremely difficult to identify isolated code snippets that exclusively use these supported statement types while also aligning with the scope and constraints of individual test levels.

Beyond that, several other challenges emerged during preliminary test case selection from real-world projects:

- Many real-world snippets do not produce clear outputs, making it difficult or impossible to validate or compare execution correctness.
- Code snippets in these projects almost always rely on externally defined variables or functions. While CodeDetective is capable of resolving such dependencies under certain conditions, CodeRunner performs no such resolution and fails prematurely. These failures are not caused by the actual conceptual challenge being tested, but rather by the lack of necessary context. This prevents a fair and direct comparison.
- CodeDetective requires type-annotated Python code, but most open-source projects do not use typing consistently. Adapting such code to meet the typing requirements introduces significant risk of introducing unintended behaviour and compromises the authenticity of the evaluation.
- Even when code snippets align with the intended focus of a test level (e.g., involving conditional logic), they frequently include additional logic or unsupported statement types. While the former complicates the isolation of specific behaviours, the latter makes it impossible to use such snippets with CodeDetective, as it only supports a restricted set of statements.

[1] `https://github.com/`
[2] `https://github.com/psf/black`
[3] `https://github.com/pallets/flask`
[4] `https://github.com/scrapy/scrapy`

To overcome these issues, the evaluation uses a synthetic benchmark project[5] specifically created for this thesis. Each test case in the benchmark is designed to align with a clearly defined difficulty level and is fully compatible with the technical requirements of CodeDetective. This approach allows for:

- All test cases are built using only the statement types that CodeDetective supports, avoiding execution failures caused by unsupported constructs.
- Every test case produces explicit output, making it possible to validate results and compute accuracy and success metrics reliably.
- Each test case focuses on a single concept or challenge, such as various forms of `if`-statements (e.g., different conditions, presence of `elif`/`else`, nesting, partial selection, etc.), enabling more targeted and meaningful evaluation.
- The test cases are constructed in a way that provides both tools with the required context, reducing failures caused by missing inputs and allowing a fairer comparison.

6.5 Conceptual Comparison (EQ_3)

Beyond measuring execution accuracy, we analyse how CodeDetective conceptually differs from other tools. Specifically, we compare it to LExecutor and Code Runner in terms of:

- **Dependency resolution**: Whether the tool resolves dependencies.
- **Handling missing values**: Whether the tool can handle missing values.
- **Type annotation requirement**: Whether the tool requires explicit type annotations.
- **Support incomplete code**: Whether the tool can work on code snippets as input project.
- **Control missing values**: Whether the tool provides mechanisms to let the user control the value of missing values.
- **Idempotency**: Whether repeated executions yield the same output under identical conditions.
- **Multi-language support**: Whether the tool is limited to Python or supports multiple programming languages.

[5] `https://github.com/Luiise/benchmark_project`

CHAPTER 7

Results and Discussion

7.1 Synthetic Benchmark Project and Test Cases

The evaluation is based on a synthetic benchmark project[1] specifically created to support the assessment of the proposed slicing approach realized by CodeDetective. The benchmark was designed to satisfy both the constraints of CodeDetective and the requirements associated with the test levels. The project simulates a simplified access control system composed of multiple interrelated modules and diverse Python constructs. Each module contributes a specific functional aspect to the system, such as user modeling, access evaluation, diagnostics, logging, and network interaction. The codebase is illustrative and constructed to include only supported syntax elements, such as annotated assignments, conditionals, class and function definitions, imports, and selected expressions. Additionally, it uses strictly typed Python, as required by CodeDetective. The benchmark project is set up with a local `venv`, following common practices in Python development. It provides a controlled environment for testing isolated code regions across a wide range of slicing challenges.

The test cases are grouped into ten difficulty levels, as described in 6.2, each targeting a specific slicing challenge such as function dependencies, module imports, control flow, or user-provided input. To enable observable and verifiable behaviour, each test case concludes with a print statement, allowing the produced output to be compared to the expected result. Each test case is executed independently using CodeDetective.

For each difficulty level, this section demonstrates a representative test case that reflects the slicing challenge described in Section 6.2. A complete summary of all test cases, including file locations, line ranges, required inputs, and expected outputs, is provided in Appendix A, along with the full code snippets used in the evaluation.

[1] `https://github.com/Luiise/benchmark_project`

7.1.1 Level 1 – Basic Expressions without Dependencies

Level 1 provides an initial evaluation of the slicing system by examining self-contained expressions that do not rely on external variables or function calls. These test cases are independent from other code elements and require no user input or dependency resolution.

A representative example, test case 1.3 is shown below:

```
    print("[" + "create".upper() + "]" + " log entry")
```

This test case includes a string method call combined with concatenation. It demonstrates that the slicer correctly retains isolated expressions even when involving basic string operations.

7.1.2 Level 2 – Variable Dependencies and Expressions within Snippet

Level 2 evaluates whether the slicing system correctly resolves simple intra-function dependencies and performs expression evaluation over intermediate variables.

A representative example, test case 2.3:

```
    startup_ms: int = 1325
    threshold: int = 1000
    is_slow_boot: bool = startup_ms > threshold
    print("Slow boot detected:", is_slow_boot)
```

This example uses internal variables and a boolean expression before producing output. It tests the ability of CodeDetective to retain all relevant statements that contribute to a computed result.

7.1.3 Level 3 – Handling Conditional Statements

Level 3 focuses on conditional structures and their resolution, including `if-elif-else` constructs, nested conditions, and various condition expressions.

A representative example, test case 3.1:

```
    role: str = "admin"
    clearance: int = 2

    if role == "admin":
        if clearance >= 3:
            print("Access level: Full admin")
        else:
            print("Access level: Partial admin")
    elif role == "guest":
        print("Access level: Guest")
    else:
        print("Access level: Unknown")
```

This example features a full nested conditional structure. It demonstrates that the slicing process must account for branching logic and only include the paths required by the region under inspection.

7.1.4 Level 4 – Functions (Simple & Complex)

Level 4 focuses on test cases involving function calls across the program. It includes chained calls, functions with arguments, return values, and recursion. These cases assess whether the slicing process correctly follows interprocedural dependencies.

A representative example, test case 4.3:

```
    formatted: str = format_name("Alice")
    print("Welcome,", formatted)
```

This test case includes a call to a helper function and a subsequent use of its return value. It evaluates the ability od CodeDetective to include the relevant function definition and preserve interprocedural dependencies.

7.1.5 Level 5 – Imports (Standard, Third-Party, Project Files)

Level 5 evaluates the ability of CodeDetective to handle symbols imported from various sources, including local modules, standard libraries and third-party packages. These cases test resolution of external dependencies outside the current source file.

A representative example, test case 5.3:

```
    user_id: str = "ID-001"
    is_valid: bool = re.match(r"ID-\d{3}", user_id) is not None
    print("Valid admin ID format:", is_valid)
```

This test case makes use of the `re` module from the standard library. It demonstrates whether imports are properly resolved and retained when used in an expression.

7.1.6 Level 6 – Object-Oriented Constructs

Level 6 evaluates object-oriented features including class instantiation, inheritance, method calls and attribute access. These test cases verify whether slicing preserves class structure and related behaviour.

A representative example, test case 6.2:

```
    guest: Guest = Guest("Bob")
    guest.grant_temp_code("ABC123")
    print("Guest has access:", guest.has_access())
```

This case involves object instantiation and method calls across a class hierarchy. It checks whether the slicing process can correctly track class-based behaviour and include necessary class and method definitions.

7.1.7 Level 7 – User Input (Primitive Types)

Level 7 contains test cases that depend on unresolved variables of primitive types (e.g., `int`, `str`, `bool`, `float`). These inputs are expected to be provided by the user during slicing and type-checked accordingly.

A representative example, test case 7.2:

```
    if role == "admin" and is_verified and (login_hour >= 8 and login_hour <=
     18):
            print("Admin access granted during business hours")
```

This snippet references multiple primitive variables which are not defined locally. It evaluates whether the tool correctly identifies these as unresolved inputs and integrates user-provided values.

7.1.8 Level 8 – User Input (Complex Types)

Level 8 introduces user-provided variables of complex types, including `List`, `Dict`, `Set`, `Union`, `Literal` and custom class instances. These cases evaluate how the system handles structured and type-constrained input values.

A representative example, test case 8.3:

```
    label: str = describe_user(user)
    if user.is_admin:
        print(label + " has elevated rights")
    print(label + " has limited rights")
```

This test case depends on a structured user object with attributes accessed both directly and via a helper function. It verifies whether custom object input and associated behaviour is supported by CodeDetective.

7.1.9 Level 9 – API Interactions

Level 9 focuses on code that performs HTTP-based API interactions using either the standard library or third-party modules. These test cases involve sending requests, decoding responses, and producing observable output.

A representative example, test case 9.2:

```
    url: str = f"https://jsonplaceholder.typicode.com/users/{user_id}"
    response: HTTPResponse = urllib.request.urlopen(url)
    body: str = response.read().decode("utf-8")
    data: dict = json.loads(body)

    if local_status == "denied":
        print("Access denied and logged for", data["username"])
    else:
        print("Access granted and logged for", data["username"])
```

This snippet demonstrates dynamic URL construction, external API access, and conditional handling of data. It evaluates whether slicing supports network interactions and response parsing.

7.1.10 Level 10 – File and Database Access

Level 10 focuses on test cases that perform persistent data operations, including file I/O and database writes. This level evaluates the ability of CodeDetective to retain functional behaviour across file paths, database backends, and structured writes.

A representative example, test case 10.1:

```
    f: object = open(path, "r")
    content: str = f.read()
    print("File content:", content)
```

This example involves reading from a file path stored in a variable and printing the result. It assesses whether I/O operations are preserved correctly, even when path variables are externally defined.

7.2 EQ_1 - CodeDetective Accuracy

EQ_1 evaluates the accuracy of CodeDetective by assessing whether the output produced by executing the generated program slices matches the expected result. Across all test cases, CodeDetective achieved a correctness rate of 82.69%. Figure 7.1 visualizes the accuracy per level, while Table 7.1 provides a numerical breakdown and lists the failing test cases.

Level	Total Cases	Correct Cases	Failing Cases	Correctness Rate (%)
1	5	5		100.00
2	5	5		100.00
3	5	4	3.4	80.00
4	5	4	4.1	80.00
5	5	4	5.5	80.00
6	5	4	6.5	80.00
7	5	5		100.00
8	5	5		100.00
9	5	3	9.4, 9.5	60.00
10	7	4	10.1, 10.4, 10.6	57.14

Table 7.1: EQ_1: CodeDetective accuracy score per test level.

CodeDetective achieved perfect accuracy in Levels 1, 2, 7, and 8. These levels test basic expressions, both with and without intra-snippet dependencies and include user-provided values of primitive and structured types. The results confirm that CodeDetective correctly

preserves independent or self-contained statements, accurately resolves local variable dependencies and successfully integrates missing values through validated user input. Structured types such as objects, lists, dictionaries, and literals are handled correctly. This demonstrates that the input prompts and reconstruction logic operate reliably even for nested structures.

For Levels 3 through 6, CodeDetective correctly executed four out of five test cases per level, each achieving a correctness rate of 80%.

Level 3 focuses on conditional control flow through `if`/`elif`/`else` structures. CodeDetective correctly executed four of the five cases, showing that it can handle conditional branches and include relevant clauses based on dependency analysis. The failure in test case 3.4 resulted from a structurally incomplete selection, where only an `elif` clause was included without its preceding `if` block, causing a `SyntaxError`. This failure highlights a general limitation in handling structurally incomplete input regions, where selected lines do not form a syntactically valid program segment. The slicing process is designed to preserve relevant logic without attempting to repair invalid input selections.

Level 4 includes function definitions and interprocedural calls. CodeDetective succeeded in most cases by resolving dependencies that span multiple functions and correctly reconstructing call relationships. Only test case 4.1 produced unexpected output. In this case, the region of interest consisted solely of a function definition, which was correctly extracted. However, the slicer mistakenly duplicated the internal body statements as additional top-level code in the reconstructed file. As a result, these statements were executed unintentionally when the program was run, producing unexpected output.

Level 5 verifies that CodeDetective correctly resolves and tracks project-local function and variable imports, as well as standard library usages. In cases involving third-party dependencies not available in the slicing environment, however, execution fails with a module import error. This was the case for test case 5.5, which required a package not preinstalled in the environment of CodeDetective. However, such test cases can be resolved by manually installing the required dependencies, which confirms that the failure is not due to slicing logic but rather to environmental configuration.

Level 6 focuses on class-based object-oriented constructs. In four of the five test cases, CodeDetective handled these constructs successfully, including correct instantiation of objects, resolution of attributes and method calls, and proper handling of class inheritance. The failure in test case 6.5 occurred because the selected region was located within a class method, but the slicing process did not include the surrounding class definition. As a result, references to `self` could not be resolved, causing execution to fail.

Level 9 achieved a correctness rate of 60.00% and focuses on code that interacts with external APIs. In three of the five test cases, CodeDetective successfully reconstructed the required logic for issuing HTTP requests and processing responses. These results confirm that basic network communication and response handling are structurally supported by the slicing mechanism. The same limitation affecting third-party libraries reappears in test cases 9.4 and 9.5, both of which use the external HTTP client `httpx`. If this library

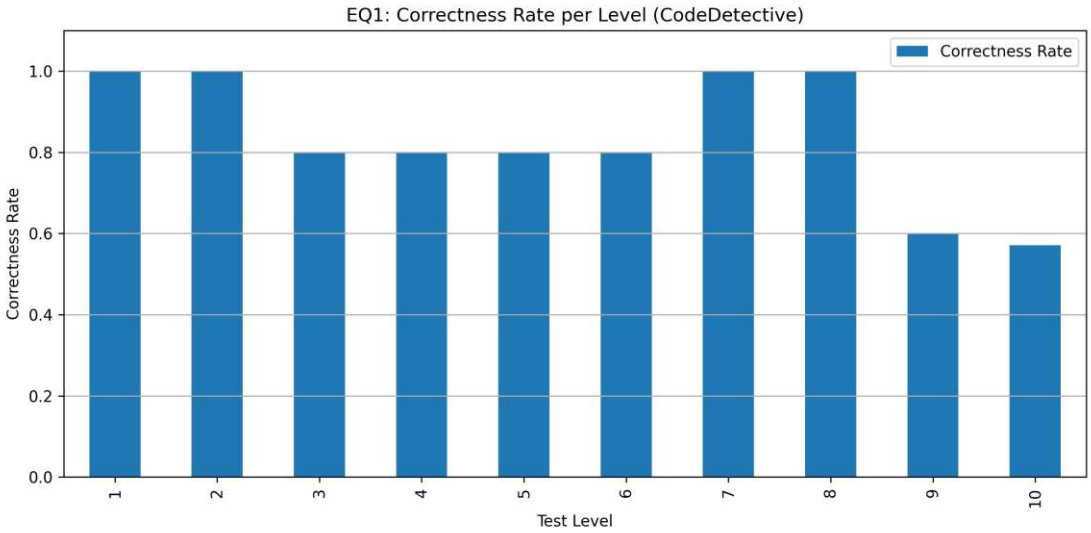

Figure 7.1: EQ_1: CodeDetective correctness rate per test level.

is unavailable during slice execution, a `ModuleNotFoundError` occurs. Nevertheless, these test cases run correctly when `httpx` is installed, showing that API requests and response handling are structurally supported by the slicing logic.

Level 10 covered both file I/O and database operations and achieved an accuracy score of 57.14%, with results varying depending on how paths and connections were managed. Test cases involving absolute paths or explicit resource setup executed correctly and confirmed that CodeDetective supports persistent data access. In contrast, test cases using relative file paths failed because the sliced code runs in a new context where such paths are no longer resolvable. Execution of database operations depends on the availability of a configured database and complete connection logic within the selected region. Test case 10.4 failed due to a missing table in an SQLite database, while test case 10.5 succeeded using identical logic with proper setup. The PostgreSQL test case 10.6 failed due to an unavailable driver library, but 10.7, which used the same logic in a properly configured environment, executed successfully. Hence, these outcomes demonstrate that external resource handling is supported when dependencies and setup are included or satisfied externally.

7.3 EQ_2 - Performance Comparison

The aim of EQ_2 is to compare the performance, in terms of success rate and accuracy, of CodeDetective against Code Runner (CR). Code Runner is a Visual Studio Code extension which enables developers to execute selected code snippets. While it facilitates code execution, it lacks advanced program analysis capabilities. Specifically, it does not resolve dependencies within the selected code, instead merely copying the code snippet

into a new temporary file for execution. In many real-world scenarios, however, code snippets are embedded within function or class definitions and therefore reside in local rather than global scopes. As a result, directly executing the copied code often leads to `IndentationError`.

To explore whether this structural issue is the primary source of failure, an additional baseline configuration called CR-global is considered. By placing the snippet in the global scope, CR-global enables execution of code that would otherwise fail due to indentation, while keeping the snippet itself unchanged. It does not provide dependency resolution or input handling, but mitigates structural issues introduced by nested scopes.

The results of this comparative evaluation are summarized in Table 7.2. CodeDetective achieved a correctness rate of 82.69% and a success rate of 84.61%, considerably outperforming both CR and CR-global. Original CR reached only 9.62% in both metrics, failing in nearly all test cases beyond Level 1. The CR-global variant achieved 28.85% for both success and correctness, showing limited improvement in early levels, but no significant gains beyond Level 3, particularly when dependencies or missing inputs are involved.

Figure 7.2 and Figure 7.3 provide a visual comparison of success and correctness rates across all levels. These bar charts illustrate the same performance trends observed in the table, highlighting the sharp contrast between CodeDetective and the baseline approaches.

Level	**CD-Succ** (%)	**CD-Corr** (%)	**CR-Succ** (%)	**CR-Corr** (%)	**CRG-Succ** (%)	**CRG-Corr** (%)
1	100.00	100.00	80.00	80.00	100.00	100.00
2	100.00	100.00	00.00	00.00	100.00	100.00
3	80.00	80.00	00.00	00.00	80.00	80.00
4	80.00	100.00	20.00	20.00	20.00	20.00
5	80.00	80.00	0.00	0.00	0.00	0.00
6	80.00	80.00	0.00	0.00	0.00	0.00
7	100.00	100.00	0.00	0.00	0.00	0.00
8	100.00	100.00	0.00	0.00	0.00	0.00
9	60.00	60.00	0.00	0.00	0.00	0.00
10	57.14	57.14	0.00	0.00	0.00	0.00
Avg	**84.61**	**82.69**	**9.62**	**9.62**	**28.85**	**28.85**

Table 7.2: EQ_2: Success rate (Succ) and correctness rate (Corr) comparison for CodeDetective (CD), CodeRunner (CR), and CodeRunner with global insertion (CRG).

It is worth noting that across all systems and test levels, success and correctness rate are nearly identical. A successful execution almost always produces the correct output, which confirms that when a tool is able to run the sliced code without runtime errors,

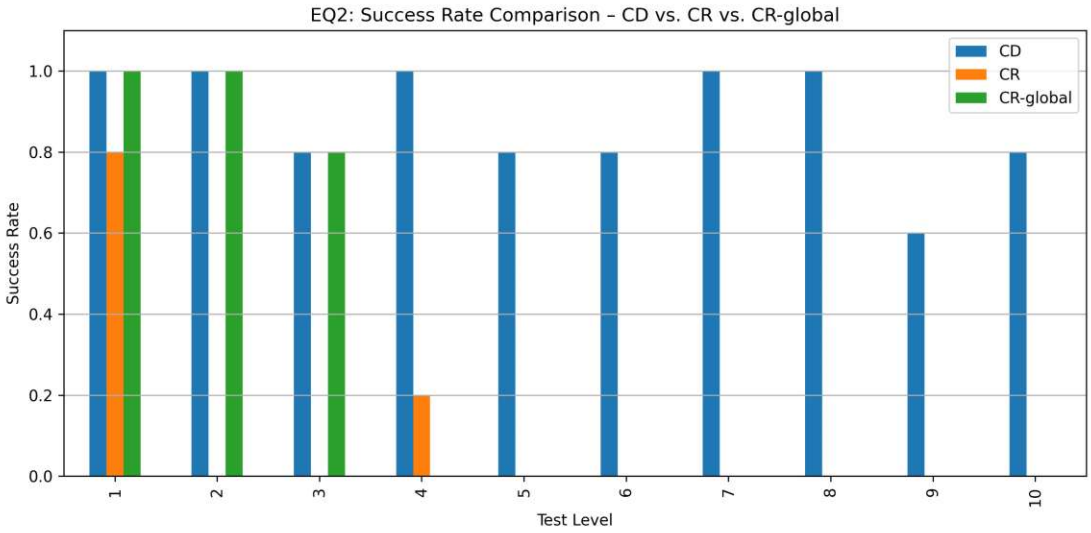

Figure 7.2: EQ_2: Execution success comparison between CodeDetective, CodeRunner and CR-global.

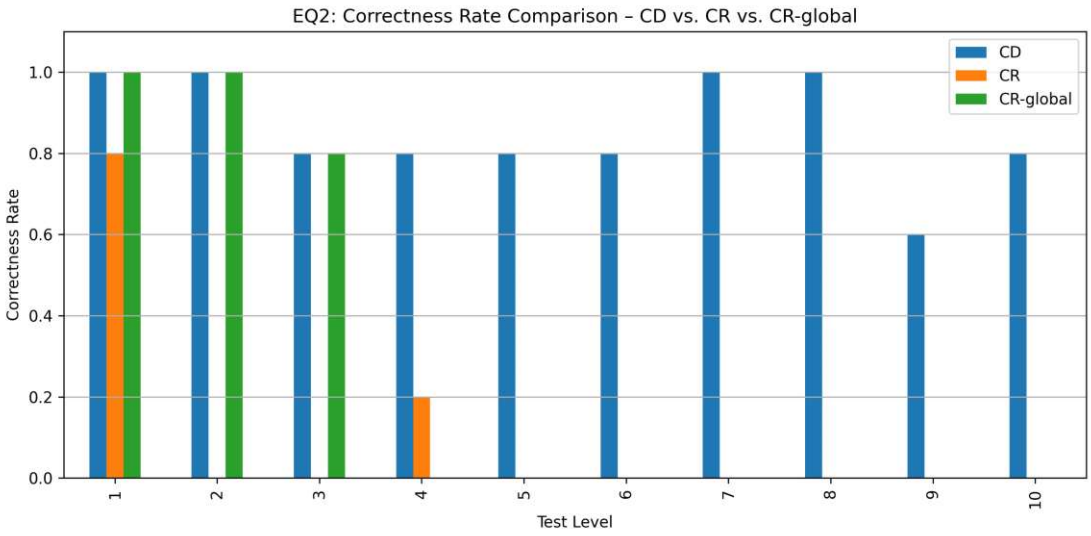

Figure 7.3: EQ_2: Correctness comparison between CodeDetective, CodeRunner and CR-global.

the behaviour is highly likely to be semantically accurate. The only exception is test case 4.1 with CodeDetective, where the selected region included a function definition, but body statements were mistakenly duplicated outside the function, causing unintended output. This was discussed in Section 7.2.

In Levels 1 and 2, both CR-global and CodeDetective perform well, while CR fails in several cases due to `IndentationError` caused by nested code structures. CR-global resolved some of these by relocating the selected region into the global scope, eliminating indentation-related failures in the early levels. This is particularly evident in Levels 2 and 3, where the correctness rate increases from 0% to 100% and 80%, respectively. These improvements result entirely from eliminating indentation issues by relocating the code, rather than from any enhanced handling of dependencies or input.

From Level 4 onward, the effectiveness of CR and CR-global declines significantly. These test cases involve interprocedural logic, imports, or unresolved variables and both baselines fail to include the necessary definitions and statements required for correct execution. CodeDetective retains correct execution in most of these scenarios by resolving dependencies beyond the selected region.

In Levels 7 and 8, where user input is required, both CR variants fail entirely. Since neither approach can prompt for input or validate types, execution fails with a `NameError` or similar exception. In contrast, CodeDetective detects unresolved values, prompts the user for valid input and ensures type consistency prior to execution, achieving full correctness in both levels.

Levels 9 and 10 further highlight the limitations of naive snippet execution in real-world programs. Code snippets interacting with APIs or persistent storage typically depend on imports, file paths, network access, or configured environments. Without a mechanism to analyse and preserve those dependencies, both CR and CR-global are unable to execute any of the relevant test cases. CodeDetective succeeds in several of these cases, showing that execution is possible when the required configuration or environment is available.

To better understand error evolution between CR and CR-global, Figure 7.4 displays a heatmap of error transitions. Figure 7.5 provides a complementary Sankey diagram. Most errors in CR are `IndentationError`, which are partially resolved in CR-global by placing the snippet at global scope. However, many of those transitions lead to new `NameError` exceptions, indicating missing dependency resultion to constructs outside the RUI. Only a limited number of cases transitioned to successful execution.

7.4 EQ_3 - Conceptual Comparison

This evaluation question investigates how CodeDetective conceptually differs from two existing tools that support the execution of code snippets: Code Runner and LExecutor [27], a research tool that enables the execution of incomplete Python code by dynamically injecting values during runtime.

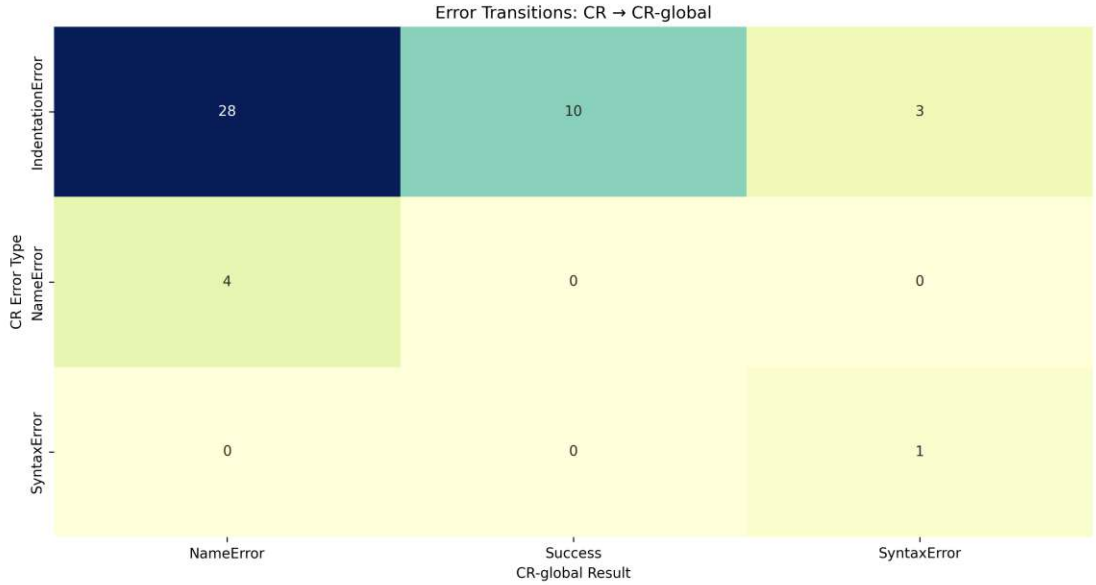

Figure 7.4: EQ_2: Heatmap showing the transition of CR errors under CR-global execution context.

The comparison focuses on a set of conceptual criteria that distinguish the underlying approaches of each tool. Table 7.3 presents which concepts are supported by each tool.

Concept	**Code Runner**	**LExecutor**	**CodeDetective**
Dependency resolution	✗	✗	✓
Handling missing values	✗	✓	✓
Requires type annotations	✗	✗	✓
Supports incomplete code	✓	✓	✗
User control over values	✗	✗	✓
Idempotent output	✓	✗	✓
Multi-language support	✓	✗	✗

Table 7.3: Conceptual Comparison of CodeDetective with LExecutor and Code Runner

CodeDetective performs static dependency resolution based on program analysis. It constructs a dependency graph from the selected code region and includes only the minimal set of statements necessary for correct execution. Code Runner does not apply any form of program analysis and instead executes the selected snippet in isolation, without resolving references to definitions outside the selected code. LExecutor does not resolve dependencies statically either, but allows execution to continue by dynamically injecting values when missing definitions are encountered.

In terms of handling missing values, both LExecutor and CodeDetective enable execution

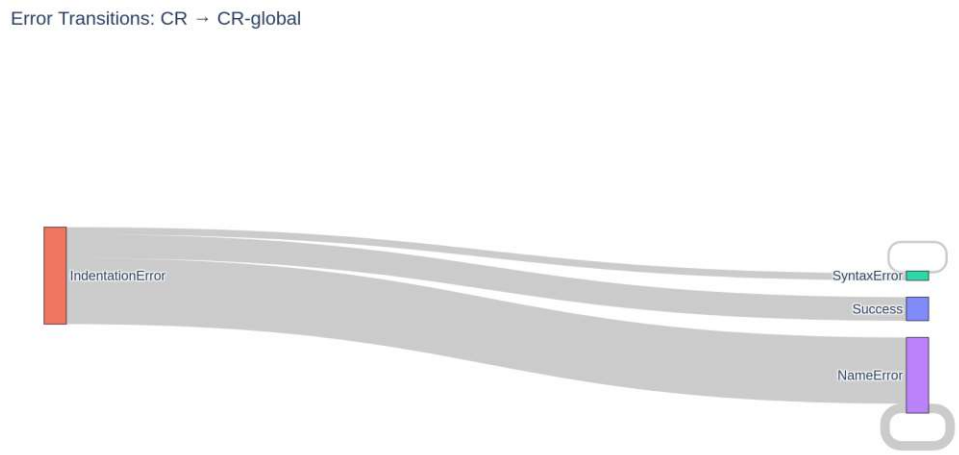

Figure 7.5: EQ_2: Sankey diagram showing the transition of CR errors under CR-global execution context.

in the presence of undefined symbols. LExecutor handles this by intercepting runtime errors and predicting suitable values using a trained neural model. CodeDetective prompts the user for concrete values and validates them against inferred types. Code Runner provides no mechanism for handling missing values and fails on unresolved references.

CodeDetective requires type annotations, which are used during variable tracking and to enforce well-typed user inputs. Neither Code Runner nor LExecutor require annotations.

In terms of support for incomplete code, both LExecutor and Code Runner are more flexible. They are capable of executing partial code snippets even when no complete project context is available. CodeDetective, on the other hand, requires a complete and syntactically correct typed program as input. It assumes that all referenced components can be analysed and resolved using static analysis.

Only CodeDetective provides control over unresolved values. During execution, users are prompted to provide concrete inputs that match the expected types, enabling controlled experimentation and reproducibility. LExecutor synthesizes values automatically based on model predictions, while Code Runner offers no input resolution mechanism at all.

The tools also differ in their execution consistency, particularly with respect to idempotent behaviour. Both Code Runner and CodeDetective produce consistent results for the same input, assuming no side effects. LExecutor, however, does not guarantee idempotent output, as its prediction mechanism may yield different results across runs due to

stochastic variation in the generated values.

Multi-language support is offered only by Code Runner, which supports several major programming languages including Python, Java, JavaScript, and C++. LExecutor is implemented specifically for Python and relies on language-specific instrumentation and prediction models. CodeDetective is designed specifically for statically typed Python programs and does not currently support other languages as well.

CHAPTER 8

Conclusion and Future Work

This thesis proposed, CodeDetective, a slicing-based system for executing isolated regions of Python code. The tool combines program slicing, dependency resolution, user interaction, and code regeneration to enable seamless execution of code snippets outside their original context. By reconstructing only the necessary dependencies and prompting for unresolved input values, CodeDetective allows developers to explore arbitrary regions of code without relying on the full execution context of the original program.

The central research questions guiding this work were concerned with both the feasibility of building such a system and its effectiveness in practice.

- **RQ1** How can we build a program analyser that produces a subset of executable code snippets from a source file?
- **RQ2** How well does the developed tool perform in evaluating code snippets of varying complexity and structure within controlled testing environments?

Regarding **RQ1**, this thesis presents an approach that derives an executable subset $P_S \subseteq P$ from a selected region of code, preserving its intended behaviour while enabling isolated execution. The method performs recursive dependency analysis, which resolves both direct and transitive dependencies by traversing the program and analysing control and data flow relationships. A variable tracking mechanism is used to distinguish resolvable from unresolved values. Missing inputs are presented to the user through type-validated prompts, allowing them to provide the necessary values interactively before execution. After all necessary elements have been identified, the extracted slice is reconstructed into a standalone script using topological sorting and structured rewriting. This process preserves the behaviour of the selected region while allowing the resulting code to execute independently of its original surrounding context.

RQ2 investigated how well the tool performs on a range of code snippets with increasing complexity. To evaluate this, a benchmark suite consisting of ten conceptual difficulty levels was constructed, covering scenarios from simple expressions to advanced cases involving missing inputs and external effects. The results showed that CodeDetective achieves a correctness rate of over 80% across all test cases, significantly outperforming baseline approaches such as CodeRunner and its global variant, which only reached nearly 10% and 30% respectively. In contrast to the baseline methods, CodeDetective succeeded in executing a wide range of code regions that depended on additional context, including function calls, user-provided values, or external side effects. These findings confirm that dependency reconstruction and interactive variable resolution are critical for enabling execution of code snippets that rely on contextual information beyond the selected region.

While CodeDetective demonstrates strong performance across a variety of test scenarios, several limitations suggest promising directions for future work. The current implementation of CodeDetective is limited to a core subset of Python constructs handled by specific statement handlers. Extending support to cover a broader range of language features, such as loops and exception handling, would allow it to operate on more representative and complex programs.

Additionally, CodeDetective requires the input program to be statically typed and relies on type annotations to support variable resolution and input validation. Relaxing this constraint or introducing fallback mechanisms for type inference would improve usability in more diverse environments.

CodeDetective operates independently of the original runtime environment, so execution may fail when code depends on relative file paths or unavailable third-party packages. Future work could investigate strategies for environment-aware slicing and more robust handling of external dependencies.

Finally, integrating CodeDetective into development environments such as Visual Studio Code could improve usability. While the tool is currently used through a command-line interface, embedding it into an IDE would allow developers to execute slices directly from selected regions in the editor, making the tool more convenient to use during development.

In conclusion, this thesis introduced an approach to enabling isolated code execution that combines program analysis with interactive input resolution. CodeDetective enables execution of code snippets outside their original context by reconstructing dependencies and prompting for unresolved values. While the tool already handles a wide range of scenarios, future work may extend its language coverage, improve environmental awareness and enhance accessibility through IDE integration.

APPENDIX A

Code Snippets of Benchmark Test Cases

This appendix provides a full overview of the benchmark test cases used for evaluation. Each level is presented with:

- A table summarizing all test cases (including file, line range, concept, user input, and expected output)
- The corresponding code snippet for each test case

Test cases are grouped by their difficulty level.

Level 1 – Basic Expressions without Dependencies

Table A.1 summarizes the test cases in this level.

The corresponding code snippets are shown below.

Test Case 1.1 (`startup.py`, line 3)

```
1 print("System is about to start.")
```

Test Case 1.2 (`startup.py`, line 6)

```
1     print("=" * 3 + " ACCESS SYSTEM v1.0 " + "=" * 3)
```

Test Case 1.3 (`utils.py`, line 17)

```
1     print("[" + "create".upper() + "]" + " log entry")
```

ID	File	Line	Concept	Expected Output
1.1	`startup.py`	3	Simple single print statement	`System is about to start.`
1.2	`startup.py`	6	Single print statement with string multiplication and concatenation	`=== ACCESS SYSTEM v1.0 ===`
1.3	`utils.py`	17	Print with string method and concatenation	`[CREATE]log entry`

1.4	`access_control.py`	9–13	Triple-quoted multi-line string	`+--------------------+`
				`| Access Control Initialized |`
				`+--------------------+`

1.5	`startup.py`	13–15	Consecutive independent print statements	`Initializing authentication module...`
				`Loading user access profiles...`
				`Establishing database connection...`

Table A.1: Test cases for Level 1 – Basic Expressions without Dependencies

Test Case 1.4 (`access_control.py`, lines 9–13)

```
    print("""
    +----------------------------+
    |  Access Control Initialized |
    +----------------------------+
    """)
```

Test Case 1.5 (`startup.py`, lines 13–15)

```
    print("Initializing authentication module...")
    print("Loading user access profiles...")
    print("Establishing database connection...")
```

Level 2 – Variable Dependencies and Expressions within Snippet

Table A.2 summarizes the test cases in this level.

The corresponding code snippets are shown below.

Test Case 2.1 (`startup.py`, line 20–22)

ID	File	Line	Concept	Expected Output
2.1	`startup.py`	20-22	Independent assignment + sequential print using that value	`System ready. Welcome to the system`
2.2	`startup.py`	12-17	String transformation and use across two variables	`Initializing authentication module... Loading user access profiles... Establishing database connection... Done: BOOT COMPLETE`
2.3	`system_info.py`	8-11	Internal variable dependency and boolean comparison	`Slow boot detected: True`
2.4	`system_info.py`	14-17	Boolean comparison across defined constants	`Admin quorum reached: True`
2.5	`startup.py`	32-34	Arithmetic and type conversion	`Boot completed in 1.385 seconds.`

Table A.2: Test cases for Level 2 – Variable Dependencies and Expressions

```
    welcome_message: str = "Welcome to the system"
    print(f"System ready.")
    print(f"{welcome_message}")
```

Test Case 2.2 (`startup.py`, line 12–17)

```
    startup_message: str = "boot complete".upper()
    print("Initializing authentication module...")
    print("Loading user access profiles...")
    print("Establishing database connection...")
    boot_msg: str = f"Done: {startup_message}"
    print(boot_msg)
```

Test Case 2.3 (`system_info.py`, line 8–11)

```
    startup_ms: int = 1325
    threshold: int = 1000
    is_slow_boot: bool = startup_ms > threshold
    print("Slow boot detected:", is_slow_boot)
```

Test Case 2.4 (`system_info`, lines 14–17)

```
    current_admins: int = 3
    required: int = 2
```

```
    has_enough_admins: bool = current_admins >= required
    print("Admin quorum reached:", has_enough_admins)
```

Test Case 2.5 (`startup.py`, lines 32–34)

```
    boot_time_ms: int = 1385
    seconds: float = boot_time_ms / 1000
    print("Boot completed in", seconds, "seconds.")
```

Level 3 – Handling Conditional Statements

Table A.3 summarizes the test cases in this level.

ID	File	Line	Concept	Expected Output
3.1	`access_control.py`	26-37	Full if/elif/else structure with nested if	`Access level: Partial admin`
3.2	`access_control.py`	26-31	Outer and inner if branch only	–
3.3	`access_control.py`	26-33	Full outer if branch with full inner if	`Access level: Partial admin`
3.4	`access_control.py`	34-35	Isolated elif branch	–
3.5	`access_control.py`	41-46	Complex condition	`Admin access granted during business hours`

Table A.3: Test cases for Level 3 – Handling Conditional Statements

The corresponding code snippets are shown below.

Test Case 3.1 (`access_control.py`, lines 26–37)

```
    role: str = "admin"
    clearance: int = 2

    if role == "admin":
        if clearance >= 3:
            print("Access level: Full admin")
        else:
            print("Access level: Partial admin")
    elif role == "guest":
        print("Access level: Guest")
    else:
        print("Access level: Unknown")
```

Test Case 3.2 (`access_control.py`, lines 26–31)

```
    role: str = "admin"
    clearance: int = 2

    if role == "admin":
        if clearance >= 3:
            print("Access level: Full admin")
```

Test Case 3.3 (`access_control.py`, lines 26–33)

```
    role: str = "admin"
    clearance: int = 2

    if role == "admin":
        if clearance >= 3:
            print("Access level: Full admin")
        else:
            print("Access level: Partial admin")
```

Test Case 3.4 (`access_control.py`, lines 34–35)

```
    elif role == "guest":
        print("Access level: Guest")
```

Test Case 3.5 (`access_control.py`, lines 41–46)

```
    role: str = "admin"
    is_verified: bool = True
    login_hour: int = 9

    if role == "admin" and is_verified and (login_hour >= 8 and login_hour <=
     18):
        print("Admin access granted during business hours")
```

Level 4 – Functions (Simple & Complex)

Table A.4 summarizes the test cases in this level.

The corresponding code snippets are shown below.

Test Case 4.1 (`startup.py`, lines 11–17)

```
def boot_message() -> str:
    startup_message: str = "boot complete".upper()
    print("Initializing authentication module...")
    print("Loading user access profiles...")
    print("Establishing database connection...")
    boot_msg: str = f"Done: {startup_message}"
    print(boot_msg)
```

ID	File	Line	Concept	Expected Output
4.1	`startup.py`	11-17	Simple function without arguments	–
4.2	`startup.py`	38	Function chaining	`=== ACCESS SYSTEM v1.0 === Initializing authentication module... Loading user access profiles... Establishing database connection... Done: BOOT COMPLETE`
4.3	`user_greeting.py`	6-7	Function call with argument	`Welcome, [USER] Alice`
4.4	`system_info.py`	23-25	Function call with usage of return value	`== System uptime: 5h 32m ==`
4.5	`startup.py`	46-47	Recursive function call	`Startup in: 3 2 1 0`

Table A.4: Test cases for Level 4 – Functions (Simple & Complex)

Test Case 4.2 (`startup.py`, line 38)

```
1    start_system()
```

Test Case 4.3 (`user_greeting.py`, lines 6–7)

```
1    formatted: str = format_name("Alice")
2    print("Welcome,", formatted)
```

Test Case 4.4 (`system_info.py`, lines 23–25)

```
1    message: str = get_uptime_message()
2    decorated: str = "== " + message + " =="
3    print(decorated)
```

Test Case 4.5 (`startup.py`, lines 46–47)

```
1    result: str = count_down(3)
2    print("Startup in:", result)
```

Level 5 – Imports (Standard, Third-Party, Project Files)

Table A.5 summarizes the test cases in this level.

ID	File	Line	Concept	Expected Output
5.1	`system_info.py`	28	Use of imported project-local function	`System ready: Ready`
5.2	`report.py`	22-24	Use of imported project-local variable	`[LOG] Alice: login`
5.3	`utils.py`	28-30	Use of standard library regex	`Valid admin ID format: True`
5.4	`report.py`	27-29	Use of standard library `math` function	`Reported CPU load: 72 %`
5.5	`report.py`	16	Use of third-party library	`FORBIDDEN!`

Table A.5: Test cases for Level 5 – Imports (Standard, Third-Party, Project Files)

The corresponding code snippets are shown below.

Test Case 5.1 (`system_info.py`, line 28)

```
    print("System ready: ", system_ready)
```

Test Case 5.2 (`report.py`, lines 22–24)

```
    username: str = "Alice"
    action: str = "login"
    print(format_log(username,action))
```

Test Case 5.3 (`utils.py`, lines 28–30)

```
    user_id: str = "ID-001"
    is_valid: bool = re.match(r"ID-d{3}", user_id) is not None
    print("Valid admin ID format:", is_valid)
```

Test Case 5.4 (`report.py`, lines 27–29)

```
    load_percent: float = 71.3
    rounded: int = math.ceil(load_percent)
    print("Reported CPU load:", rounded, "%")
```

Test Case 5.5 (`report.py`, line 16)

```
    print(Fore.RED + "forbidden!".upper())
```

Level 6 – Object-Oriented Constructs

Table A.6 summarizes the test cases in this level.

ID	File	Line	Concept	Expected Output
6.1	`system_info.py`	32-33	Object instantiation and method call	`User has access: True`
6.2	`system_info.py`	36-38	Subclass instantiation, state mutation, and method call	`Guest has access: True`
6.3	`utils.py`	36-38	Object passed as argument and accessed inside a helper function	`Formatted user label: [USER] Clara`
6.4	`utils.py`	41-42	Method call defined in a subclass	`Admin role: Admin - cyber security`
6.5	`user.py`	13	Attribute access through self inside a class method	`nan`

Table A.6: Test cases for Level 6 – Object-Oriented Constructs

The corresponding code snippets are shown below.

Test Case 6.1 (`system_info.py`, lines 32–33)

```
    user: User = User("Alice", is_admin=True)
    print("User has access:", user.has_access())
```

Test Case 6.2 (`system_info.py`, lines 36–38)

```
    guest: Guest = Guest("Bob")
    guest.grant_temp_code("ABC123")
    print("Guest has access:", guest.has_access())
```

Test Case 6.3 (`utils.py`, lines 36–38)

```
    user: User = User("Clara", is_admin=False)
    label: str = get_user_label(user)
    print("Formatted user label:", label)
```

Test Case 6.4 (`utils.py`, lines 41–42)

```
    admin: Admin = Admin("Alice", department="cyber security")
    print("Admin role:", admin.get_role())
```

Test Case 6.5 (`user.py`, line 13)

```
    print("Access granted to:", self.user)
```

Level 7 – User Input (Primitive Types)

Table A.7 summarizes the test cases in this level.

ID	File	Line	Concept	User Input	Expected Output
7.1	`startup.py`	16-17	Requires user variable of type string	`startup_message: "Hello World!"`	`Done: Hello World!`
7.2	`access_control.py`	45-46	Requires multiple user variables (string, float, bool)	`is_verified = True, login_hour = 18, role = "admin"`	`Admin access granted during business hours`
7.3	`report.py`	11-19	Requires user variable only defined in one if branch	`code = 404, ok_str = "okay"`	`Unknown status`
7.4	`report.py`	35-37	Requires user variable of type float, uses math library	`cpu = 12.4, memory = 38.9`	`Total system usage (rounded): 52`
7.5	`access_control.py`	29-37	Requires user variable of type string and int	`role = "guest", clearance = 10`	`Access level: Guest`

Table A.7: Test cases for Level 7 – User Input (Primitive Types)

The corresponding code snippets are shown below.

Test Case 7.1 (`startup.py`, lines 16–17)

```
    boot_msg: str = f"Done: {startup_message}"
    print(boot_msg)
```

Test Case 7.2 (`access_control.py`, lines 45–46)

```
    if role == "admin" and is_verified and (login_hour >= 8 and login_hour <=
     18):
        print("Admin access granted during business hours")
```

Test Case 7.3 (`report.py`, lines 11–19)

```
    if code == 200:
        ok_str = "OK"
        print(ok_str)
        return ok_str
```

Test Case 7.4 (`report.py`, lines 35–37)

```
    total: float = cpu + memory
    rounded: int = math.ceil(total)
    print("Total system usage (rounded):", rounded, "%")
```

Test Case 7.5 (`access_control.py`, lines 29–37)

```
    if role == "admin":
        if clearance >= 3:
            print("Access level: Full admin")
        else:
            print("Access level: Partial admin")
    elif role == "guest":
        print("Access level: Guest")
    else:
        print("Access level: Unknown")
```

Level 8 – User Input (Complex Types)

Table A.8 summarizes the test cases in this level.

The corresponding code snippets are shown below.

Test Case 8.1 (`access_control.py`, lines 55–60)

```
    if status == "granted":
        print(f"Notification: Access granted for {username}")
    elif status == "denied":
        print(f"Notification: Access denied for {username}")
    else:
        print(f"Notification: Unknown status for {username}")
```

Test Case 8.2 (`access_control.py`, lines 63–66)

```
    if isinstance(roles, str):
        print("Single access role:", roles)
    else:
        print("Multiple access roles:", ", ".join(roles))
```

Test Case 8.3 (`access_control.py`, lines 18–22)

```
    label: str = describe_user(user)
    if user.is_admin:
        print(label + " has elevated rights")
    print(label + " has limited rights")
```

Test Case 8.4 (`report.py`, lines 40–41)

```
    top_user: User = max(access_counts, key=access_counts.get)
    print("Top user:", top_user.username)
```

Test Case 8.5 (`report.py`, lines 44–47)

```
    if user in allowed_users:
        print("Access granted for", user.username)
    else:
        print("Access denied for", user.username)
```

Level 9 – API Interactions

Table A.9 summarizes the test cases in this level.

The corresponding code snippets are shown below.

Test Case 9.1 (`newtwork.py`, lines 10–14)

```
    url: str = f"https://jsonplaceholder.typicode.com/users/{user_id}"
    response: HTTPResponse = urllib.request.urlopen(url)
    body: str = response.read().decode("utf-8")
    data: dict = json.loads(body)
    print("User email:", data["email"])
```

Test Case 9.2 (`newtwork.py`, lines 18–26)

```
    url: str = f"https://jsonplaceholder.typicode.com/users/{user_id}"
    response: HTTPResponse = urllib.request.urlopen(url)
    body: str = response.read().decode("utf-8")
    data: dict = json.loads(body)

    if local_status == "denied":
        print("Access denied and logged for", data["username"])
    else:
        print("Access granted and logged for", data["username"])
```

Test Case 9.3 (`newtwork.py`, lines 30–36)

```
    url: str = "https://httpbin.org/post"
    payload: dict = {"user": "admin", "action": "login"}
    data: bytes = json.dumps(payload).encode("utf-8")
    request = urllib.request.Request(url, data=data, method="POST", headers={
    "Content-Type": "application/json"})
    response = urllib.request.urlopen(request)
    body: str = response.read().decode("utf-8")
    result: dict = json.loads(body)
    print("POSTed user:", result["json"]["user"])
```

Test Case 9.4 (`newtwork.py`, lines 46–48)

```
    response: Response = httpx.get("https://httpbin.org/get")
    result: dict = response.json()
    print("Your User-Agent was:", result["headers"]["User-Agent"])
```

Test Case 9.5 (`newtwork.py`, lines 40–42)

```
    payload: dict = {"user": "admin", "event": "login"}
    response = httpx.post("https://httpbin.org/post", json=payload)
    print("Third-party POST status code:", response.status_code)
```

Level 10 – File and Database Access

Table A.10 summarizes the test cases in this level.

The corresponding code snippets are shown below.

Test Case 10.1 (`utils.py`, lines 45–47)

```
    f: object = open(path, "r")
    content: str = f.read()
    print("File content:", content)
```

Test Case 10.2 (`utils.py`, lines 45–47)

```
    f: object = open(path, "r")
    content: str = f.read()
    print("File content:", content)
```

Test Case 10.3 (`utils.py`, lines 45–47)

```
    f: object = open(path, "r")
    content: str = f.read()
    print("File content:", content)
```

Test Case 10.4 (`storage_sqlite.py`, lines 7–14)

```
    conn: sqlite3.Connection = sqlite3.connect(DB_PATH)
    cursor: sqlite3.Cursor = conn.cursor()
    inserted: int = cursor.execute(
        "INSERT INTO access_log (username, status) VALUES (?, ?)",
        (username, status)
    ).rowcount
    conn.commit()
    print("Inserted rows:", inserted)
```

Test Case 10.5 (`storage_sqlite.py`, lines 7–14)

```
    conn: sqlite3.Connection = sqlite3.connect(DB_PATH)
    cursor: sqlite3.Cursor = conn.cursor()
    inserted: int = cursor.execute(
        "INSERT INTO access_log (username, status) VALUES (?, ?)",
        (username, status)
    ).rowcount
    conn.commit()
    print("Inserted rows:", inserted)
```

Test Case 10.6 (`storage_pg.py`, lines 11–18)

```
    conn: psycopg2.extensions.connection = psycopg2.connect(dbname=DB_NAME,
    user=DB_USER, password=DB_PASS, host=DB_HOST, port=DB_PORT)
    cursor: psycopg2.extensions.cursor = conn.cursor()
    inserted: int = cursor.execute(
        "INSERT INTO access_log (username, status) VALUES (%s, %s)",
        (username, status)
    )
    conn.commit()
    print("Inserted access event into PostgreSQL.")
```

Test Case 10.7 (`storage_pg.py`, lines 11–18)

```
    conn: psycopg2.extensions.connection = psycopg2.connect(dbname=DB_NAME,
    user=DB_USER, password=DB_PASS, host=DB_HOST, port=DB_PORT)
    cursor: psycopg2.extensions.cursor = conn.cursor()
    inserted: int = cursor.execute(
        "INSERT INTO access_log (username, status) VALUES (%s, %s)",
        (username, status)
    )
    conn.commit()
    print("Inserted access event into PostgreSQL.")
```

ID	File	Line	Concept	User Input	Expected Output
8.1	access_control.py	55-60	Requires user variables of type string and Literal	`status = 'granted', username = "Alice"`	`Notification: Access granted for Alice`
8.2	access_control.py	63-66	Requires user variable of Union type	`roles = ["admin", "developer", "user"]`	`Multiple access roles: admin, developer, user`
8.3	access_control.py	18-22	Requires user variable of custom type	`user = User(username = "Alice", is_admin = True)`	`Alice (admin) has elevated rights`
8.4	report.py	40-41	Requires user variable of dict type with custom class as key	`access_counts = {User(is_admin= True, username= 'Alice'): 100, User(is_admin= False, username= 'Bob'): 20}`	`Top user: Alice`
8.5	report.py	44-47	Requires user variable of set type with custom class	`allowed_users = [User( username="Alice", is_admin=True), User(username= "Bob", is_admin=False), User( username="clara", is_admin=False)] user = User( username= "Mallory", is_admin=False)`	`Access granted for Alice`

Table A.8: Test cases for Level 8 – User Input (Complex Types)

ID	File	Line	Concept	User Input	Expected Output
9.1	`newtwork.py`	10-14	External API call with user variable using standard library	`user_id = 3`	`User email: Nathan@yesenia.net`
9.2	`newtwork.py`	18-26	External API call with conditional response handling	`user_id = 2, local_status = "denied"`	`Access denied and logged for Antonette`
9.3	`newtwork.py`	30-36	External POST API call using standard library	–	`POSTed user: admin`
9.4	`newtwork.py`	46-48	GET request using third-party library	–	`Your User-Agent was: python-httpx/0.27.0`
9.5	`newtwork.py`	40-42	POST request using third-party library	–	`Third-party POST status code: 200`

Table A.9: Test cases for Level 9 – API Interactions

ID	File	Line	Concept	User Input	Expected Output
10.1	`utils.py`	45-47	Basic file write using internal file	`logs/ access_log.txt`	`File content: [INFO] alice logged in [INFO] bob accessed secure file [WARN] guest denied access`
10.2	`utils.py`	45-47	Write to a fully resolved internal path	`/home/luise/Programs/benchmark_project/logs/access_log.txt`	`File content: [INFO] alice logged in [INFO] bob accessed secure file [WARN] guest denied access`
10.3	`utils.py`	45-47	External file system write (outside project boundary)	`/tmp/output.txt`	`File content: [INFO] alice logged in [INFO] bob accessed secure file [WARN] guest denied access`
10.4	`storage_sqlite.py`	7-14	File-based relational DB write using sqlite3 without defined DB	`username = "Alice", status = "logged out"`	`Inserted access event into PostgreSQL.`
10.5	`storage_sqlite.py`	7-14	File-based relational DB write using sqlite3 with definedn DB and Table	`username = "Alice", status = "logged out"`	`Inserted access event into PostgreSQL.`
10.6	`storage_pg.py`	11-18	Remote DB write using psycopg3 without database setup	`username = "Alice", status = "logged out"`	`Inserted access event into PostgreSQL.`
10.7	`storage_pg.py`	11-18	Remote DB write using psycopg3 with database setup	`username = "Alice", status = "logged out"`	`Inserted access event into PostgreSQL.`

Table A.10: Test cases for Level 10 – File and Database Access

Overview of Generative AI Tools Used

1. DeepL[1]
   - **Usage**: Used to translate the abstract and acknowledgements.
   - **Place of use**: Abstract, acknowledgements.
2. ChatGPT[2]
   - **Usage**: Used to rephrase sentences in order to improve readability and clarity.
   - **Place of use**: Entire thesis.

[1] https://www.deepl.com/en/translator
[2] https://openai.com/index/chatgpt/

List of Figures

List of Tables

List of Algorithms

Acronyms

AST Abstract Syntax Tree. 9, 10, 20, 22, 24, 25, 47–49

CR Code Runner. 63

DFS Depth-first search. 11, 12, 95

DG Dependency Graph. 20

LLM Large Language Models. 47

PDG Program Dependence Graph. 15

PyCG Practical Python Call Graphs. 13, 14

R2E Repository to Environment. 47

RUI Range Under Investigation. 17, 20, 24–27, 33–35, 37, 44, 48, 51, 66

VRT Variable Resolution Tracker. 20, 24, 27

VUT Variable Usage Tracker. 49

Bibliography

[1] ast — Abstract Syntax Trees — docs.python.org. https://docs.python.org/3/library/ast.html. [Accessed 26-03-2025].

[2] URL https://frama-c.com/index.html.

[3] inspect — Inspect live objects. https://docs.python.org/3/library/inspect.html. [Accessed 16-04-2024].

[4] Python sys.settrace. https://docs.python.org/3/library/sys.html#sys.settrace. [Accessed 16-04-2024].

[5] Topological Sorting - GeeksforGeeks — geeksforgeeks.org. https://www.geeksforgeeks.org/topological-sorting/. [Accessed 28-03-2025].

[6] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006. ISBN 0321486811.

[7] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The dogged pursuit of bug-free c programs: the frama-c software analysis platform. *Commun. ACM*, 64(8):56–68, jul 2021. ISSN 0001-0782. doi: 10.1145/3470569. URL https://doi.org/10.1145/3470569.

[8] Kasra Ferdowsi, Ruanqianqian Huang, Michael B. James, Nadia Polikarpova, and Sorin Lerner. Validating ai-generated code with live programming. *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2023. URL https://api.semanticscholar.org/CorpusID:259187909.

[9] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987. ISSN 0164-0925. doi: 10.1145/24039.24041. URL https://doi.org/10.1145/24039.24041.

[10] Markus Gaelli. Modeling examples to test and understand software. 2006. URL https://api.semanticscholar.org/CorpusID:69252359.

[11] Patrice Godefroid. Micro execution. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 539–549, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568273. URL https://doi.org/10.1145/2568225.2568273.

[12] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. *SIGPLAN Not.*, 32(10):108–124, October 1997. ISSN 0362-1340. doi: 10.1145/263700.264352. URL https://doi.org/10.1145/263700.264352.

[13] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, page 108–124, New York, NY, USA, 1997. Association for Computing Machinery. ISBN 0897919084. doi: 10.1145/263698.264352. URL https://doi.org/10.1145/263698.264352.

[14] Prabhat Gupta. Building a rule engine with abstract syntax trees (AST) in java: A comprehensive guide. https://www.nected.ai/blog/rule-engine-with-abstract-syntax-trees-ast-in-java, December 2024. Accessed: 2025-3-26.

[15] Sebastian Haug, Christoph Böhm, and Daniel Mayer. Automated code generation and validation for software components of microcontrollers, 2025. URL https://arxiv.org/abs/2502.18905.

[16] Andre Hora. Spotflow: Tracking method calls and states at runtime. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion '24, page 35–39, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705021. doi: 10.1145/3639478.3640029. URL https://doi.org/10.1145/3639478.3640029.

[17] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, page 35–46, New York, NY, USA, 1988. Association for Computing Machinery. ISBN 0897912691. doi: 10.1145/53990.53994. URL https://doi.org/10.1145/53990.53994.

[18] Zeyuan Hu. Graph basics + topological sort, Jul 2018. URL https://zhu45.org/posts/2018/Jul/14/graph-basics-topological-sort/.

[19] Naman Jain, Manish Shetty, Tianjun Zhang, King Han, Koushik Sen, and Ion Stoica. R2e: Turning any github repository into a programming agent environment. In *ICML*, 2024.

[20] Han Jun. Code runner. URL https://github.com/formulahendry/vscode-code-runner?tab=readme-ov-file#code-runner.

[21] Steve D. Lazaro. A return into the world of static analysis with frama-c, Feb 2014. URL http://www.mupuf.org/blog/2014/02/10/a_return_into_the_world_of_frama-c/.

[22] Sorin Lerner. Projection boxes: On-the-fly reconfigurable visualization for live programming. *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020. URL https://api.semanticscholar.org/CorpusID:211138019.

[23] Ondrej Lhoták. Comparing call graphs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, page 37–42, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595935953. doi: 10.1145/1251535.1251542. URL https://doi.org/10.1145/1251535.1251542.

[24] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. On the comprehension of program comprehension. *ACM Trans. Softw. Eng. Methodol.*, 23(4), sep 2014. ISSN 1049-331X. doi: 10.1145/2622669. URL https://doi.org/10.1145/2622669.

[25] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the First ACM SIGSOFT-/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, page 177–184, New York, NY, USA, 1984. Association for Computing Machinery. ISBN 0897911318. doi: 10.1145/800020.808263. URL https://doi.org/10.1145/800020.808263.

[26] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. Pycg: Practical call graph generation in python. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, page 1646–1657. IEEE Press, 2021. ISBN 9781450390859. doi: 10.1109/ICSE43902.2021.00146. URL https://doi.org/10.1109/ICSE43902.2021.00146.

[27] Beatriz Souza and Michael Pradel. Lexecutor: Learning-guided execution. *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023. URL https://api.semanticscholar.org/CorpusID:256615851.

[28] Bastian Steinert, Michael Perscheid, Martin Beck, Jens Lincke, and Robert Hirschfeld. Debugging into examples. In Manuel Núñez, Paul Baker, and Mercedes G. Merayo, editors, *Testing of Software and Communication Systems*, pages 235–240, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-05031-2.

[29] Karthik Chandra Swarna, Noble Saji Mathews, Dheeraj Vagavolu, and Sridhar Chimalakonda. On the impact of multiple source code representations on software engineering tasks — an empirical study. *J. Syst. Softw.*, 210(C), April 2024. ISSN

0164-1212. doi: 10.1016/j.jss.2023.111941. URL https://doi.org/10.1016/j.jss.2023.111941.

[30] Frank Tip. A survey of program slicing techniques. Technical report, NLD, 1994.

[31] Vijay Walunj, Gharib Gharibi, Duy H. Ho, and Yugyung Lee. Graphevo: Characterizing and understanding software evolution using call graphs. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 4799–4807, 2019. doi: 10.1109/BigData47090.2019.9005560.

[32] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10 (4):352–357, 1984. doi: 10.1109/TSE.1984.5010248.

[33] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, March 2005. ISSN 0163-5948. doi: 10.1145/1050849.1050865. URL https://doi.org/10.1145/1050849.1050865.

[34] Litao Yan, Alyssa Hwang, Zhiyuan Wu, and Andrew Head. Ivie: Lightweight anchored explanations of just-generated code. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, CHI '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703300. doi: 10.1145/3613904.3642239. URL https://doi.org/10.1145/3613904.3642239.