B.Comp. Dissertation

Automating Software Debugging: An Approach to Travel Back to The Root Cause of Your Bug

By

Hongshu Wang

Department of Computer Science

School of Computing

National University of Singapore

2023/2024

Project No.: H0111950

Supervisor: Dr. Dong Jinsong

Evaluator: Dr. Boyd Anderson

B.Comp. Dissertation

Automating Software Debugging: An Approach to Travel Back to The Root Cause of Your Bug

By
Hongshu Wang

Department of Computer Science
School of Computing
National University of Singapore

2023/2024

Project No.: H0111950
Supervisor: Dr. Dong Jinsong
Evaluator: Dr. Boyd Anderson

Abstract

Software debugging is known to be a tedious, challenging, yet ineluctable process. This project aims to accomplish a research work that automates software debugging through time-traveling approach. Various existing approaches such as feedback-based debugging and probabilistic inference were studied. In these approaches, either excessive human intervention or computational resources is required, limiting their efficiency in minimizing debugging efforts. To overcome these issues, a solution called DebugPilot was proposed. The contributions of this Final Year Project are on the IO Detection for simulation experiments and formula refinement.

In addition, this project identifies and aims to solve the problem of missing data dependencies. Data dependency analysis is a crucial step in code analysis tasks. Various research has been conducted with the assumption of the availability of a complete data flow. However, in practice, this assumption might not always be true, and there is a need for a technique to ensure the completeness of data flows. This project aims to address the incomplete data flow problem in backtracking-based software debugging by developing a technique to recover missing data dependencies. By solving this problem, the number of usable programs in Defects4J is increased. Experiments are conducted to evaluate the effectiveness of the proposed solution, and limitations are identified to provide insights into directions of future research.

Subject Descriptors

- Software and its engineering > Software creation and management > Software verification and validation > Software defect analysis > Software testing and debugging

Keywords

Fault Localization, Time-Travelling Debugging, Data Flow Analysis

Acknowledgements

I would like to express my deepest gratitude to my academic supervisors Dr. Dong Jinsong and Dr. Lin Yun for their guidance and support throughout this project.

I am also thankful to Wong Yuk Kwan and Boh Cheng Hin for their insightful discussions and support.

Finally, I would like to extend my thanks to the School of Computing in the National University of Singapore for providing the necessary resources and facilities that made this research possible.

Table of Contents

1 Introduction

1.1 Project Objectives

Software debugging is known to be tedious and challenging, especially in complex software systems, yet it is an ineluctable process in software development. This project aims to accomplish a research work that addresses this problem by automating the debugging process through a time-travelling approach. The contribution of this project should involve different aspects of the research work, such as theory refinement, experiment design, conduction, and enhancement.

In addition, this project identifies and aims to solve the problem of missing data dependencies. Data dependency analysis is a crucial step in code analysis tasks. Various research has been conducted with the assumption of the availability of a complete data flow (Lin et al., 2017; Wang et al., 2019; Guo et al., 2020; Ren et al., 2020; Wong et al., 2023). However, in practice, this assumption might not always be true, and there is a need for a technique to ensure the completeness of data flows. This project aims to address the incomplete data flow problem in backtracking-based software debugging by developing a technique to recover missing data dependencies.

1.2 Literature Review

1.2.1 Backtracking Based Software Debugging

Backtracking involves working backwards from the fault-revealing step. A technique called dynamic slicing can identify the data and control dominance relations relevant to a variable or expression (Agrawal et al., 1993), which helps in searching of the root cause in backtracking.

In 2017, Lin et al. proposed a feedback-based debugging approach, which aims to assist fault localization given a buggy program. A tool called Microbat was developed to support this work. In this approach, an execution trace with causality relations (i.e., data and control dominance relations) is generated given the buggy program. The developers are allowed to provide feedback to the trace steps, which serve as a direction of searching of the root cause along the trace. This approach has a high success rate of 92.8% in fault localization (Lin et al., 2017). However, this work has a limitation that the number of requests for feedback might be enormous, especially for complex programs with large numbers of iterations. In addition, wrong feedback might be given, but the handling is not robust enough.

Another work proposed by Xu et al. in 2018 attempted to reduce the human intervention by a probabilistic inference approach. Each executed statement is modelled as a conditional probability distribution. Information including program semantics and domain knowledge are captured by these distributions. The probability of each step being the root cause can then be calculated. One drawback of this approach is the large amount of time and resources required, which limits its effectiveness in enhancing the efficiency of the debugging process.

In 2024, Wu, S. and Yang, J. et al. proposed Devin and SWE-agent respectively. Both tools utilize natural language processing models to achieve automated debugging. Similar to the existing work, both tools require various levels of feedback as a guidance of the debugging process. For Devin, the feedback is from human, while SWE-agent predicts feedback automatically. Different from the existing work, the feedback in Devin and SWE-agent are in the form of natural language. Thus, these AI assisted debugging tools have a relatively shallow learning curve. However, the debugging approach adopted in these tools does not involve collecting execution traces. As a result, the debugging process usually requires executing the program multiple times, which is expensive when the program consumes a large amount of computational resources.

1.2.2 Data Dependency Analysis Leveraging Large Language Models

Since the deployment of large language models (LLMs) such as ChatGPT (OpenAI, 2022), research work on software engineering that incorporates the capability of LLMs has been conducted extensively (Fan et al., 2023).

In 2023, Ma et al. studied the capability of LLMs on understanding code syntax and semantics. The performance of LLMs on completing a series of code analysis tasks, including data dependency analysis, was examined. Specifically, given a segment of code, the task is to determine whether two variables are “data-dependent”. It was found that compared to CodeLLama and StarCoder, ChatGPT has higher F1 score. When comparing ChatGPT 4 with ChatGPT 3.5, it was found that they have similar performance. While this work has provided an approach of utilizing LLMs as classifiers, a large number of queries is required to build a complete data flow in a program. This is because this approach will query LLMs for each pair of variables in the program.

1.3 Overview of Microbat

1.3.1 Introduction

Microbat is a feedback-based debugging tool that was proposed in 2017 by Lin et al. Based on this tool, Tregression (Wang et al., 2019) and DebugPilot (Wong et al., 2023) were built. The tools share the same trace generation algorithm. Improvement on the trace generation would benefit all the tools. The contributions of the Final Year Project in trace generation have been described in Section 2.

1.3.2 Trace Generation

To generate an execution trace, execution information first needs to be collected. This is achieved by a technique called Java Instrumentation (java.lang.instrument package), which modifies the bytecode executed without modifying the source code. The execution information such as variable values and execution sequence can be collected by injecting instructions into the bytecode. After this, an execution trace can be generated.

1.4 Overview of DebugPilot

1.4.1 Introduction

To address the problems of excessive human intervention during debugging and low tolerance of false positives in a more computationally efficient manner, a bug-locating solution called DebugPilot was proposed (Wong et al., 2023). The contributions of the Final Year Project in this research work have been described in Section 3.

1.4.2 Debugging Plan

In this approach, a possible debugging process of the given buggy program is simulated and provided as the debugging plan. DebugPilot is an extension of the traditional time-travelling approach. Therefore, the debugging plan is composed of a series of execution steps from the fault-revealing step to the root cause. The construction involves choosing the most suspicious step until the predicted root cause is reached. The debugging plan can then be refined based on human intervention, such as correcting predicted feedback. The updated debugging plan predicts the root cause with higher accuracy because of DebugPilot's capability of learning from the developer's knowledge.

1.4.3 Computation of Suspiciousness

A heuristic called suspiciousness is introduced to determine the direction of search. Suspiciousness is a measure of the extent to which a step, variable or relation is likely to lead to the root cause. The suspiciousness of steps is initialized based on the computational cost. After this, the suspiciousness of variables and relations can be calculated based on the suspiciousness that has been computed. A debugging plan can then be generated. Similarly, the suspiciousness scores are updated upon feedback correction.

2 Data Dependency Recovery

2.1 Definitions

2.1.1 Notations

$\boldsymbol{s_i}$: a step on an execution trace with order i.

$\boldsymbol{v}$: a variable involved in the execution of the program.

$\boldsymbol{v.f}$: a field in variable v. A field is also a variable.

$\boldsymbol{r(s_i)}$: the set of read variables at step s_i.

$\boldsymbol{w(s_i)}$: the set of written variables at step s_i.

$\boldsymbol{DD_v(s_j, s_i)}$: the data dependency through variable v between step s_j and s_i, where

$$v \in r(s_j) \cap v \in w(s_i) \cap v \notin w(s_k) \text{ } where \text{ } i < k < j$$

2.1.2 Read and Written Variables

Read variable: Variable v is said to be a read variable of step s if its value is read and used at this step.

Written variable: Variable v is said to be a written variable of step s if it is assigned a value at this step.

2.1.3 Data Domination

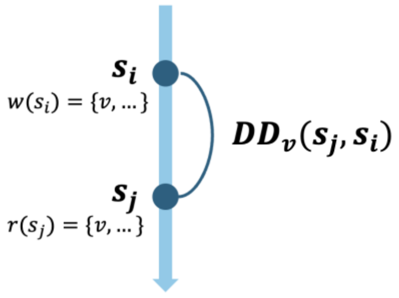


Figure 1. Data Dependency between Two Steps

Consider two steps s_i and s_j on an execution trace, where $i < j$, i.e., s_i is executed before s_j. Let v be a variable that is read by step s_j and written by step s_i, and v is not written by any step s_k between s_i and s_j. That is,

$$v \in r(s_j) \cap v \in w(s_i) \cap v \notin w(s_k) \text{ } where \text{ } i < k < j$$

The relationships between s_i, s_j and v are illustrated in Figure 1.

Data Dominator: s_i is said to be the data dominator of s_j regarding variable v.

Data Dominatee: s_j is said to be the data dominatee of s_i regarding variable v.

Data Dependency: there exists a data dependency between s_j and s_i through variable v.

2.1.4 Critical Variable

Consider the scenario when a variable v is not written by a step s, but there exists a variable $v.f$, an internal field of v, that is written by step s. Then $v.f$ is defined to be the critical variable of v at step s. That is, $\exists$ variable v with field $v.f$, at step s, $v.f$ is critical iff

$$v \notin w(s) \cap v.f \in w(s)$$

2.2 Problem Statement

2.2.1 Background

Java Instrumentation Scope

The first step in backtracking-based debugging is execution trace generation. This procedure is equivalent to setting a breakpoint at each execution step and recording the execution information. In Microbat and DebugPilot, the trace is collected through Java instrumentation. Due to time and memory constraints, only the program of interest is instrumented, any JDK classes and third-party libraries are skipped during instrumentation. Therefore, the execution information within the libraries is missing.

Selective Variable Recording

On a generated trace, instead of referring to variables by addresses, the values of variables are recorded at each step as part of the execution information. In practice, variables with composite types usually involve multiple fields, which could be further expanded, and the memory consumption increases exponentially with the number of layers of variables recorded. As a result, it is infeasible to duplicate variables completely during trace generation. Instead, only the first few layers of fields within a variable are recorded onto the trace.

2.2.2 Data Dependency Missing Problem

Let s_{root_cause} and $s_{problem\_revealing}$ represent the root cause step and the problem revealing step in a buggy program. The backtracking-based debugging approach starts from $s_{problem\_revealing}$ and searches for s_{root_cause} through the dynamic slicing algorithm

(Agrawal et al., 1993). The searching process uses execution steps as nodes, data and control dependencies as edges. As illustrated in Figure 2, missing data dependencies can break the path from $s_{problem\_revealing}$ to s_{root_cause}, and consequently lead to failure in locating the root cause.

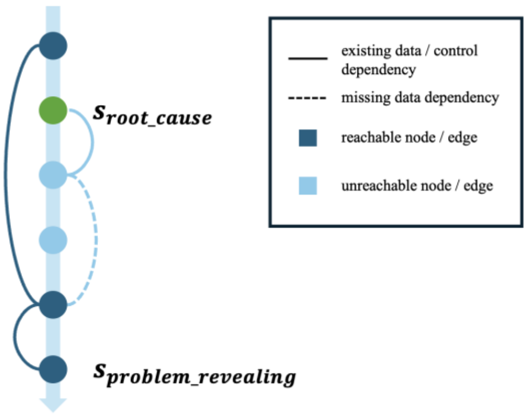


Figure 2. Data Dependency Missing Problem

Both incomplete instrumentation and partial recording of variable information serve as sources of the missing data dependency problem.

Incomplete Instrumentation

Since classes in third-party libraries are not instrumented, the execution information of library method invocation is not recorded. Consequently, the data dependencies relevant to these method invocations are missing.

In Example 1, since ArrayList is a Java built-in class, it is not instrumented. The execution details of the add and get methods are not recorded, and the method invocations are represented as one step each. Assume the number of variable layers is 1, the expected read and written variables at each step are shown in Table 1. There exists a data dependency $DD_{l1.elementData[0]}(s_3, s_2)$, which allows the user to track the most recent step before s_3 that writes $l1.elementData[0]$. However, according to the program, there is no direct write operation at step s_2 and $l1.elementData[0] \notin w(s_2)$. This violates the definition of $DD_{l1.elementData[0]}(s_3, s_2)$, so this data dependency is missing in the generated trace.

```
s1    ArrayList < String > l1 = new ArrayList <> ( );
s2    l1.add("s");
s3    String s = l1.get(0);
```

Example 1. Data Dependency Missing Example

	$r(s)$	$w(s)$
s_1	*return value of* "new ArrayList <> ( )"	$l1$
s_2	$l1$	$l1.elementData[0]$ ▨
		$l1.size$ ▨
s_3	$l1$	s
	return value of "l1.get(0)" $\rightarrow l1.elementData[0]$	

Note: variables shaded in grey are not recorded on the trace.

Table 1. Read and Written Variables in Example 1 (Number of Variable Layers: 1)

Partial Recording of Variable Information

During trace generation, only the first few layers of fields within a variable are recorded. With limited information about the variables available, data dependency of the inner fields of variables are missing on the trace.

Consider the same example with the assumption that the number of variable layers is 2 and the $w(s_2)$ has been captured. The expected read and written variables at each step are shown in Table 2. One of the data dependencies is $DD_{l1.size}(s_3, s_2)$, following which enables a user to track the most recent step before s_3 that adds element(s) to or removes element(s) from $l1$. However, if the user decides to generate trace with one layer of variables, this dependency doesn't exist. This is because when one layer is considered, $l1.size \notin r(s_3)$, which violates the definition of $DD_{l1.size}(s_3, s_2)$.

	$\boldsymbol{r(s)}$	$\boldsymbol{w(s)}$
$\boldsymbol{s_1}$	l' *	$l1$
	$l'.size$	$l1.size$
	$l'.elementData[0]$	$l1.elementData[0]$
	$\ldots$	$\ldots$
$\boldsymbol{s_2}$	$l1$	$l1.elementData[0]$
	$l1.size$	
	$l1.elementData[0]$	$l1.size$
	$\ldots$	
$\boldsymbol{s_3}$	$l1$	s
	$l1.size$	
	$l1.elementData[0]$	
	$\ldots$	
	$return\ value\ of$ "l1.get(0)" $\rightarrow l1.elementData[0]$	

Note: variables shaded in grey are not recorded on the trace.

*: $l' = return\ value\ of$ "$new\ ArrayList <> ()$"

Table 2. Read and Written Variables in Example 1 (Number of Variable Layers: 2)

2.2.3 Motivation

While it is possible to locate root causes of some buggy programs with incomplete data flow information, it was found that the current approach failed for a large portion of programs in the Defects4J dataset. Furthermore, existing tools such as Microbat and DebugPilot allow a user to select the number of layers of variables to record. Due to the variation in available computational resources, different users might decide to record different number of layers of variables during trace generation. This leads to inconsistency in performance in these tools. In the scenario where a user decides to record the first layer of variables only, the accuracy of root cause locating is limited, and the effectiveness of the tools is restricted.

In order to improve the accuracy in root cause locating and the consistency in performance of the existing tools, it is necessary to develop a technique that recovers data dependency with limited and varying amount of available variable information.

2.3 Potential Solutions and Design Decisions

2.3.1 Comparison of Variables

Description of Approach

A source of the data dependency missing problem is that variables modified by third party libraries are not recorded in the written variable set at a given step. A naïve approach of solving this problem is by comparing the variables before and after a library method is invoked. Since two versions of a variable share the same address, the only way to compare them is through their values. Before a method invocation, the values of all the relevant variables should be stored. Later, these stored values will be compared with the variable values after method invocation.

Problems

A major problem with this solution is the high memory consumption associated with variable value recording. An instance of an object in Java contains fields, which could also have composite types and could be expanded further. To identify the modified fields, it is necessary to record all layers of fields within a variable, which is expensive in practice. If the first few layers of a variable are stored, there is a potential that some fields are modified but not detected. Therefore, this naïve approach is unable to fully recover missing data dependencies when memory is limited.

2.3.2 Data Flow Analysis

Description of Approach

The second attempt to solve this problem was through classic static program analysis techniques such as data flow analysis. In contrast with the traditional data flow analysis (Aho et al., 2007) which starts the analysis from the beginning of a program, this approach follows the back-to-forth direction in backtracking debugging. Similar to data flow analysis, a Program Dependence Graph (PDG) first needs to be constructed. Given output value of the program and ground truth values, data flows are approximated in the backward direction on the PDG. The domain and range of variable values are computed, and the data flows with contradiction in values will be eliminated. In the end, the data flows left are the recovered data flows. A more detailed explanation has been included in Appendix A.

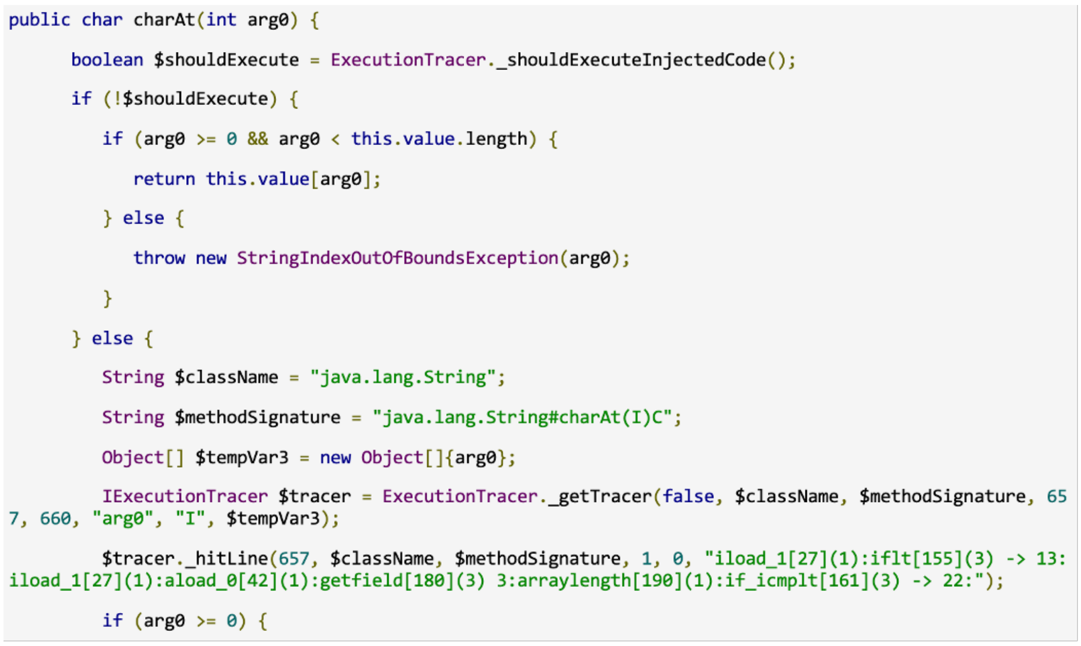
Problems

While the method described could potentially recover all the data flows in a program, there are a few problems in practice. Firstly, constructing a PDG involves analysing the source code of the whole program, which introduces additional overhead. Secondly, the process of approximating variable values on a PDG has similar time and space complexity as executing the whole program, especially when accurate data dependencies are required. As a result, this approach was not adopted, and the solution should be utilizing the execution trace.

2.3.3 Enhanced Instrumentation

Description of Approach

Another attempt to recover the data dependencies is through instrumenting the third-party libraries. In addition to the existing instrumentation algorithm, which collects execution information, a switch is added as a global variable. The status of this switch is controlled by the status of the program. The switch is on when the program starts executing, and is off when the program finishes running and the procedures for launching the UI and automated debugging start. In the libraries, the inserted instructions for collecting execution information will only be executed when the switch is on.



```
public char charAt(int arg0) {
      boolean $shouldExecute = ExecutionTracer._shouldExecuteInjectedCode();
      if (!$shouldExecute) {
         if (arg0 >= 0 && arg0 < this.value.length) {
            return this.value[arg0];
         } else {
            throw new StringIndexOutOfBoundsException(arg0);
         }
      } else {
         String $className = "java.lang.String";
         String $methodSignature = "java.lang.String#charAt(I)C";
         Object[] $tempVar3 = new Object[]{arg0};
         IExecutionTracer $tracer = ExecutionTracer._getTracer(false, $className, $methodSignature, 65
7, 660, "arg0", "I", $tempVar3);
         $tracer._hitLine(657, $className, $methodSignature, 1, 0, "iload_1[27](1):iflt[155](3) -> 13:
iload_1[27](1):aload_0[42](1):getfield[180](3) 3:arraylength[190](1):if_icmplt[161](3) -> 22:");
         if (arg0 >= 0) {
```

Figure 3. A Segment of Decompiled Code after Instrumentation

Figure 3 shows an example of enhanced instrumentation. For illustration purpose, the Java code in the figure was obtained by decompiling the bytecode after instrumentation. Note that the code segment in Figure 3 is incomplete, for a complete version please refer to Appendix B. In this example, the $charAt$ method from the java.lang.String class has been instrumented. $\$shouldExecute$ is the switch that controls whether the instrumented version should be executed.

Problems

The enhanced instrumentation approach also suffers from high computational cost. Since the instructions for collecting execution information are inserted before step in the execution, the length of bytecode after instrumentation is a few times longer than the original bytecode. This slows down the execution significantly, especially when frequently invoked JDK methods are instrumented. Another problem with this approach is that the inserted instructions are usually calling predefined static methods to collect execution information, which are also written in Java. However, since JDK methods are also instrumented, they will invoke the static methods. This causes infinite loops as the JDK methods and the information collecting methods invoke each other. Therefore, instrumenting the third-party libraries will cause stack overflow problems during execution, and this approach is unable to solve the problem.

2.4 Solution

2.4.1 Overview

As explained in section 2.2.2, the reasons of missing data dependencies at step s are incomplete sets $r(s)$ and $w(s)$. Thus, the task of recovering missing data dependencies can be simplified into identifying critical variables that are not captured during trace generation. This could be because of incomplete instrumentation and partial recording of variable information.

To address the task of identifying critical variables, a solution leveraging LLM was proposed. With the capability of LLM in understanding the code, the execution details of a library method call could be inferred. During trace generation, method invocations without instrumentation are identified. The next step in the workflow is to generate a prompt according to the format developed through prompt engineering. After the prompt is sent to a

LLM and a response is received, the last step is to understand the response and map the information to the variables on the execution trace.

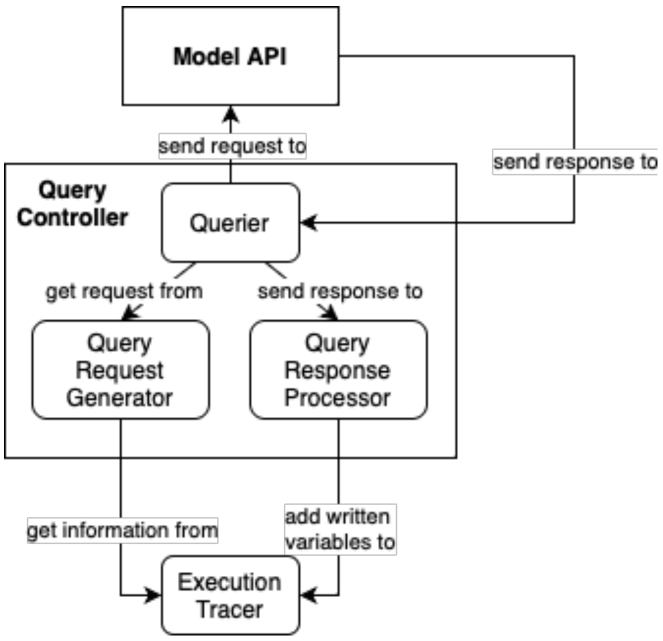


Figure 4. Architecture of Query Controller

Figure 4 shows the overall architecture of this solution. The Execution Tracer is responsible for collecting information during execution. The main contribution of this project is the Query Controller. The three components in the Query Controller correspond to the Prompt Engineering, Querying, and Variable Mapping procedures in the workflow.

2.4.2 Query Controller

Query Request Generator

The Query Request Generator is responsible for the Prompt Engineering step. It retrieves information from the Execution Tracer and uses this information to generate a LLM prompt. The main challenges for this component are the selection and retrieval of relevant information from the program, and construct a concise and informative prompt so that the LLM is able to return interpretable variable information.

Querier

The Querier is responsible for the Querying step. It acts as an interface between the Model API and the rest of the components in the Query Controller. In the proof-of-concept tool implemented as part of this FYP, ChatGPT 3.5 API was adopted. However, the architecture shown in Figure 4 supports easy migration from ChatGPT 3.5 to other LLMs. Different queriers can be implemented to communicate with different LLMs.

Query Response Processor

The Query Response Processor is responsible for the Variable Mapping step. It parses a response provided by the Querier and maps the information in the response to the variables on the execution trace. The main challenges associated with this component is the interpretation of LLM response and map the retrieved information in natural language to variables on an execution trace.

2.4.3 LLM Leveraged Data Dependency Recovery - Version 1

Observations

In Java, built-in classes like java.lang.String are immutable, which means that once created, their values cannot be modified. In contrast, mutable classes like java.util.ArrayList provide APIs to modify its internal fields. Only mutable types suffer from the missing data dependency problem. It was observed that common mutable classes usually contain an internal data structure to store elements. For example, java.util.ArrayList has an array called $elementData$ and java.util.HashMap contains an array of java.util.HashMap$Node called $table$.

According to section 2.1.4, a critical variable $v.f$ is the written field of a variable v that is not written at the outermost layer. In the examples mentioned above, the internal arrays are not directly written. Instead, their internal elements are written. Thus, the elements are the critical variables, and the internal data structure that stores these elements is referred to as a critical data structure. For example, in a java.util.ArrayList l, $l.elementData[0]$ is a candidate of critical variable since it might have been written by a step, while $l.elementData$ is a critical data structure that is not directly written.

Prompt Engineering

As mentioned in section 2.4.1, the problem of recovering missing data dependencies can be solved by identifying missing critical variables. The first version of the prompt attempts to first identify the critical data structure, then locate the critical variable by inferring a position in the critical data structure. The information about the invoked method is send to the LLM as a request, and information about the critical data structure and position containing the critical variable should be returned as a response.

Since the default LLM responses are in natural language, it is difficult to map them to variables on the trace. Therefore, a fixed format of response is defined as follows:

$$< method\ type >< method\ action >< name\ of\ internal\ array >< index >$$

The meanings and values of the tags are explained below:

$\boldsymbol{< method\ type >}$**:** Whether a line of code reads or writes values in the invoking object. It can take values from $< get/set >$. If a method reads values only, i.e. $< get >$, the tags afterward are not analyzed.

<method action>: The type of operation completed by the line of code. It can take values from $< add/remove/replace >$.

<name of internal array>: The name of the internal array used to store elements in the invoking object.

<index>: The range of positions being modified in the internal array that stores the elements. It can take values from $< start/end/all/index/key >$. Where $< start/end/index >$ represents a single position in data structures that have an ordering. $< key >$ represents a single position in data structures whose elements are retrieved by keys, such as maps and sets. $< all >$ represents that all the elements in the data structure have the potential of being written, for example, in the worst case of insertion into a priority queue, the positions of all the elements can be changed.

Note that in the implementation of an operation, there could be other positions in the critical data structure being written. For example, when adding an element to $elementData$ in java.util.ArrayList, if $elementData$ is full, more space will be allocated. However, only $elementData[i]$, where i is the index of the added element, is considered a critical variable.

To get response for an invoked method, the method signature is included in the prompt. In addition, an abstract code segment is provided. The code segment is not the actual source code, instead, it is inferred from the method signature. For example, from signature “java.util.HashMap#forEach(Ljava/util/function/BiConsumer;)V”, code segment

$hashmap.forEach(biConsumer)$ can be extracted. The question about a method is formatted as follows:

$\{abstract\ code\}\ with\ signature\ \{method\ signature\}$

For the example mentioned above, the expected output is $< set >< replace >< table >< all >$.

The prompt is composed of the explanation of the response format, some examples, and the method being queried. The format of the prompt is shown below, a complete prompt prefix with the actual examples used is included in Appendix C.

“
Return in the following format: <method type><method action><name of internal array><index>.
<method type> can take values from <get/set>, it represents whether the line of code gets or sets values in the invoking object. When a method is <get>, the tags afterward are not needed.
<method action> can take values from <add/remove/replace>, it represents the type of operation done by the line of code. When a method is <remove>, the tags afterward are not needed.
<name of internal array> represents the name of the internal array used to store elements in a data structure.
<index> can take values from <start/end/all/index/key>, they represent the range of positions being modified in the internal array that stores the elements.

For example:
{method 1} with signature {signature 1}:<response 1>,
{method 2} with signature {signature 2}:<response 2>,
…

Then
{method queried} with signature {signature queried}:
”

Variable Mapping

The following algorithm is applied to interpret query response and infer critical variables:
Given a step s that invokes a library method, where the invoke object is variable v.

1. Trim response and extract method type m_T, method action m_A, name of critical data structure $DS_{critical}$, and position pos.
2. If $m_T == GET$, terminate.
3. If $m_T == SET$

   3a. If $m_A == ADD/REMOVE$, record $\boldsymbol{v.size}$ as a critical variable.

   3b. If $pos == START/END/INDEX/KEY$, record $\boldsymbol{v.DS_{critical}[pos]}$ as a critical variable and terminate.

   3c. If $pos == ALL$, record $\boldsymbol{v.DS_{critical}[i]}$ for i in range $[0, v.DS_{critical}.length)$, terminate.

The above algorithm extracts a set of critical variables $S_{critical}$ from the response. The variables in the form of natural language are mapped to the variables on the execution trace through the following algorithm:

Given set of critical variables $S_{critical} = \{v_{NL_i} \mid i \in |S_{critical}|\}$, where v_{NL_i} is a variable with representation in natural language, and the invoke object v after method invocation.

For each v_{NL_i} in $S_{critical}$:

1. Retrieve the first level field f_{NL_i} of v_{NL_i} ("$size$" or the name of the critical data structure).
2. If the set of first level fields of v doesn't contain a field with name f_{NL_i}, continue to check $v_{NL_{i+1}}$.
3. Else, retrieve $v.f_i$, which has name f_{NL_i}.

   3a. If $f_{NL_i} ==$ "$size$", record $v.f_i$ and continue to check $v_{NL_{i+1}}$.

   3b. Else, retrieve the position pos_i from v_{NL_i}, then record $v.f_i[pos_i]$. Continue to check $v_{NL_{i+1}}$.

Performance Optimization

Since this version of the prompt involves abstract code segment instead of the actual source code, the query responses can be stored to avoid duplicate queries of the same method.

Limitations

Although the first version of the solution could recover missing critical variables for common data structures, there are other mutable classes that might not contain an array as a critical data structure. For example, java.util.LinkedList is a commonly used data structure in Java, but it uses linked instances of java.util.LinkedList$Node to store the items instead of an array. Furthermore, this prompt only supports instance methods that write to the fields of the invoke object. However, there exists class methods that write to the fields of one or more parameters. Therefore, a more general prompt should be designed.

2.4.4 LLM Leveraged Data Dependency Recovery - Version 2

Observations

Based on the limitations of Version 1, it is necessary to include the structure of the variable of interest in the prompt. This allows the LLM to learn the structure and infer the correct critical variable. In addition, the updated solution should be able to query multiple variables in the same request.

Prompt Engineering

In the second version of the prompt, a variable has the following format:

$$\{name: var\_name, type: var\_type, value: var\_value\}$$

Where var_value includes the internal fields of the variable, and each field is also a variable that follows this format. The natural language representation of the outermost variable is obtained in a recursive way. The recursion stops when the current variable has primitive type, or is the last layer of variable recorded.

The response is designed to include multiple critical variables at once:

$$< field1\_var\_name >; < field2\_var\_name >; \dots ; < fieldn\_var\_name >$$

Where each variable is represented by its variable name and the names of its ancestors. The natural language representation of a critical variable is

$$< layer\ 1\ var\_name\#layer\ 2\ var\_name\# \dots \#critical\ var\_name >$$

Note that the request and response use different ways to represent variables. In the request, a variable representation contains name, type, and its value before method invocation. On the other hand, the response contains critical variables, which could refer to a variable in the request or a field of the variable. The representation of each critical variable includes the names from the outermost layer variable to the critical variable, providing a structure to support variable mapping. Consider a java.util.ArrayList $l1$ with $elementData = [1]$. Then its natural language representation in the request is

{ name: $l1$, type: $java.util.ArrayList$, value:

[{ name: $elementData$, type: $java.lang.Object[]$, value:

[{ name: $com/mycompany/app/AppTest\{103,151\}l1 - 1.elementData[0]$,

type: $java.lang.Integer$, value: 1 };]

};

{ name: $size$, type: int, value: 1 };]

}

After invoking $l1.add(0,3)$, the following fields are written:

$$l1.elementData[0] = 3,\ l1.elementData[1] = 1,\ l1.size = 2$$

Thus, the expected output is

$< l1\#elementData\#com/mycompany/app/AppTest\{103,151\}l1 - 1.elementData[0] >$;

$< l1\#elementData\#com/mycompany/app/AppTest\{103,151\}l1 - 1.elementData[1] >$;

$< l1\#size >$

The format of the overall prompt is as follows:

"

Let {name:var_name,type:var_type,value:var_value} represent a variable. Return the fields in var_value that are modified. In your response, do not explain and return strictly in this format:<field1_var_name>;<field2_var_name>;...;<fieldn_var_name>

e.g., Given variable {example variable} After calling {code} once, the following fields of {variable name} are modified:<critical variable 1 name>;<critical variable 2 name>;…

Then
Given variables {queried variable 1} {queried variable 2} … After calling {code} once, the following fields of {queried variable 1 name}, {queried variable 2 name}, … are modified:
"

Variable Mapping

Given response in the format

$$< layer\ 1\ var\_name\#layer\ 2\ var\_name\# \ldots \#critical\ var\_name >$$

a set of critical variables $S_{critical}$ with natural language representation can be extracted. The following algorithm can then be applied to perform variable mapping:

Given a set of critical variables $S_{critical}$ with natural language representations, and a set of relevant variables $S_{relevant}$ from the execution trace.

For each variable v in $S_{relevant}$, and each critical variable $v_{critical}$ in $S_{critical}$:

1. Represent v as a tree of fields T_v, and represent $v_{critical}$ as a path within tree of fields $P_{v\_critical}$.
2. In T_v, search for $P_{v\_critical}$. Record the field at end of the path as the critical variable.

Performance Optimization

Since execution information is included in this version of the prompt and the response is specific to the request, the critical variables cannot be stored and reused. However, when a response indicates that a method doesn't write value to any fields involved, the method signature can be recorded. In the future, these stored methods will not be queried.

2.5 Evaluation

2.5.1 Defects4J Statistics

In this project, the dataset Defects4J (Just et al., 2014) is used. The performance of DebugPilot on Defects4J is illustrated in Figure 5. In total, there are 841 programs in DebugPilot.

The detailed explanation of each segment is as follows:

Debug Success: Execution trace is generated, and root cause can be located.

Debug Failure: Execution trace is generated but root cause cannot be located.

No Fault / Root Cause Detected: While searching for the root cause, the fault revealing step is the starting point, and the root cause is the target. This exception is thrown when execution trace is generated, but either the fault or the root cause cannot be detected before the debugging session starts.

Runtime Exception: The exception is thrown when the trace is not generated due to exception during the execution of the program.

Trace Length Exceeds Limit: The exception is thrown when the trace is not generated because it will exceed the limit (100,000 steps).

Timeout: The exception is thrown when the trace is not generated because it exceeds the time limit (10 minutes).

Other Exceptions: Other exceptions that cause trace generation to fail.

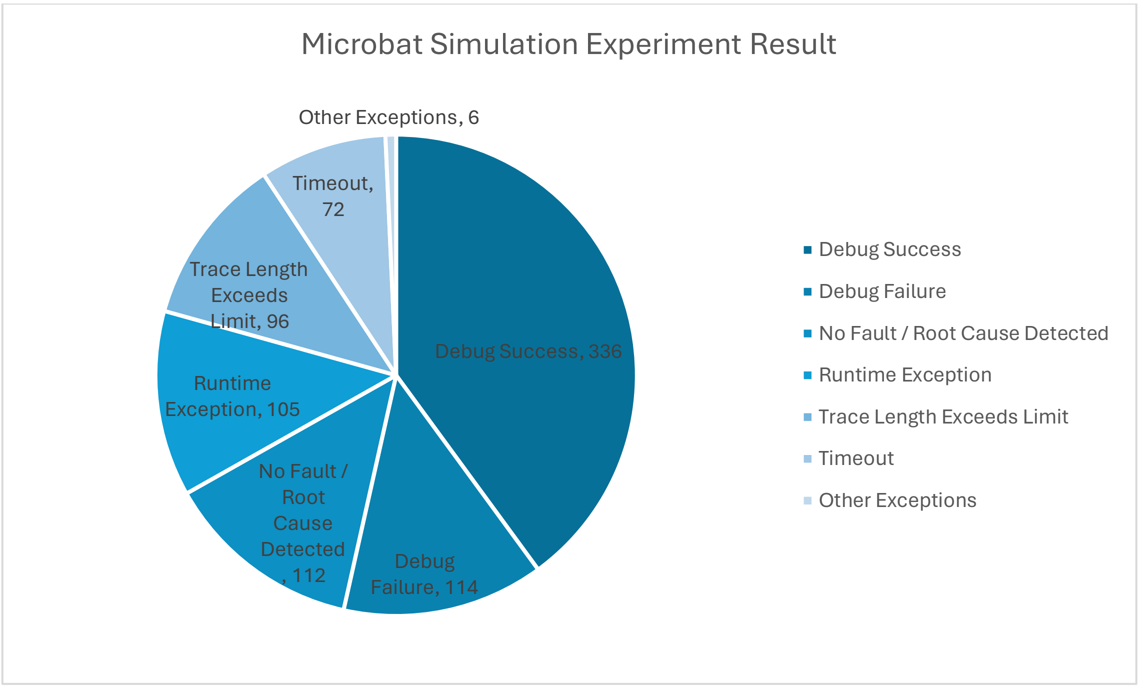


Figure 5. Microbat Simulation Experiment Result

2.5.2 Benchmark

To examine the effectiveness of the proposed solution, the critical variables of 4 representative data structures in Java are hard coded to serve as a benchmark. Each public method in these classes is hard coded with the expected output according to the response format of prompt version 1. In total, 34 methods from java.util.ArrayList, java.util.HashMap, java.util.HashSet, java.util.Queue that writes to variables were hardcoded and tested.

2.5.3 Experiment Design

Simulation experiment was conducted on the 114 programs where trace could be generated, but root cause cannot be located. Let n denote the number of variable layers. Note that among these 114 programs, the root cause cannot be located when n is 1, 2, or 3. In the simulated debugging experiment, time limit for each debugging session is set to be 10 minutes. It was found that a longer execution trace involves more LLM queries, due to time and cost considerations, the maximum length of trace is set to be 10,000 for $n = 3$ and 20,000 for $n = 1$. ChatGPT 3.5 API was used as the LLM API in this experiment.

2.5.4 Effectiveness in Addressing Missing Data Dependency due to Incomplete Instrumentation

Experiment results when n = 3 are summarized in the table below:

	Debugging Success	**Trace Generation Success**	**Percentage of Success**	**Average Trace Generation Time (s)**
Benchmark	19	93	20.43%	19.00
Prompt V1	31	88	35.23%	29.21
Prompt V2	33	65	50.77%	128.44

Table 3. Microbat Experiment Result with Data Dependency Recovery (n=3)

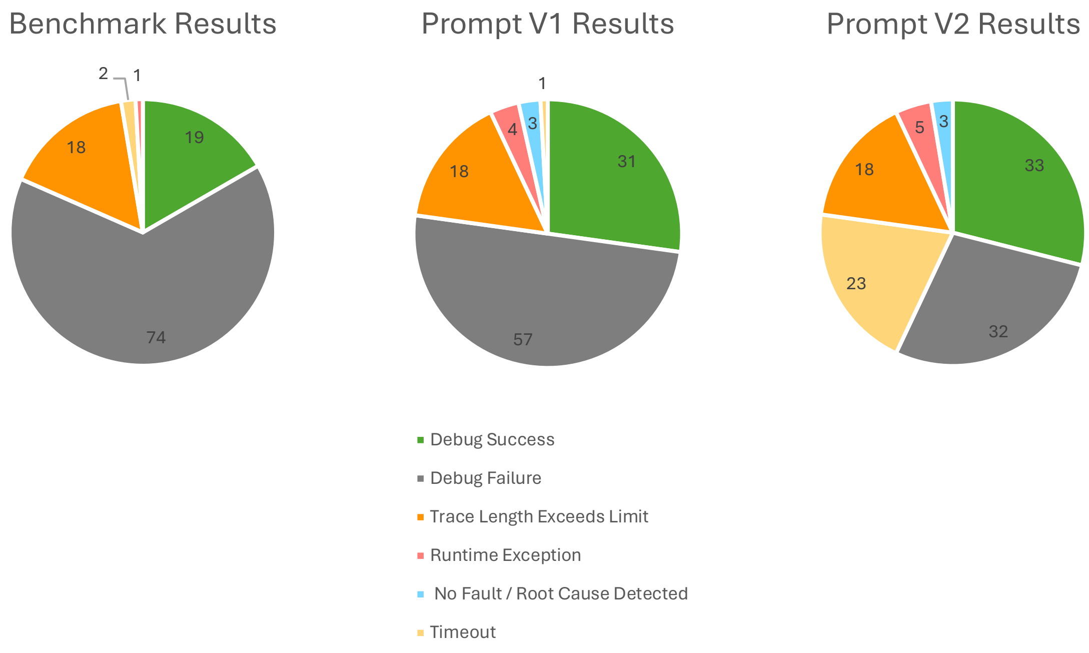


Figure 6. Microbat Experiment Result Distribution with Data Dependency Recovery (n=3)

	Debugging Success	Trace Generation Success	Percentage of Success	Average Trace Generation Time (s)
Benchmark	12	65	18.46%	15.65
Prompt V1	22	65	33.85%	15.20
Prompt V2	33	65	50.77%	128.44

Table 4. Subset of Microbat Experiment Result with Data Dependency Recovery (n=3)

As shown in Table 3, after recovering data dependencies with the proposed solution, root cause can be located in more programs in Defects4J. In addition, both versions of the solution perform better than the benchmark in number of successes in debugging and percentage of success. While Prompt V2 has similar number of successes in debugging as V1, it has a percentage of success of 50.77%.

Although Prompt V2 times out for 23 programs, among the programs where traces are successfully generated, V2 of the solution is capable of recovering more data dependencies and helping in locating root cause in more programs. This is because V2 handles class level methods as well as classes with different inner structures. As shown in Table 4, only the 65 programs that V2 can generate trace successfully are considered, and V2 has the best performance among the three groups.

The main problem with V2 is its runtime overhead. It can be seen from Table 3 and Table 4 that the runtime overhead of V1 is small compared to V2. Due to the nature of Prompt V1, the LLM response could be stored and reused. However, response for Prompt V2 are not general enough to be stored. In other words, Prompt V1 only involves static program analysis, while Prompt V2 involves execution information, which means that it adopts a dynamic program analysis approach.

2.5.5 Effectiveness in Addressing Missing Data Dependency due to Partial Recording of Variable Information

When n = 1, benchmark and prompt version 1 would not be able to recover the data dependencies, because the inner structure of the variables are not captured on the trace. Experiment results of prompt version 2 are summarized in the table below:

	Debugging Success	Trace Generation Success	Percentage of Success	Average Trace Generation Time (s)
n=1	30	88	34.09%	117.15
n=3	33	65	50.77%	128.44

Table 5. Microbat Experiment Result with Data Dependency Recovery V2 (n=1 and n=3)

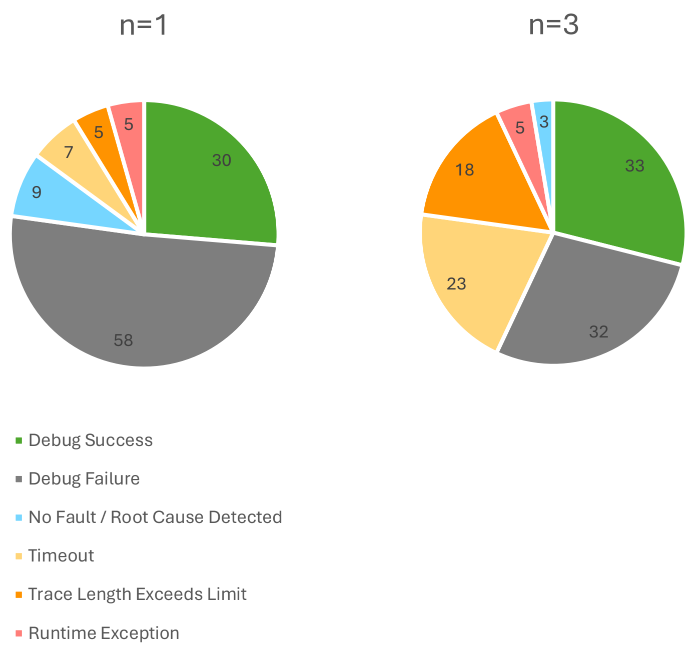


Figure 7. Microbat Experiment Result Distribution with Data Dependency Recovery V2 (n=1 and n=3)

From Table 5, it can be seen that Prompt V2 is able to recover data dependencies with limited information about the variables. By setting n to 1, the time taken for trace generation is reduced, thus more programs can be tested. However, the success rate is lower when n = 1 compared to n = 3. This could be because when less information about the variables are provided to the LLM, it is harder for the LLM to infer the inner structure of the variables. Thus, the response might not include all the critical variables.

2.6 Limitations and Future Work

As mentioned in section 2.5.4, the main drawback with the proposed solution is the runtime overhead to query a remote LLM such as ChatGPT. When the trace length increases, the debugging session also becomes expensive since a large number of ChatGPT queries is required.

The future work could focus on reducing the cost of the current solution from the perspectives of reducing the number of queries and reducing the cost of each query. To reduce the number of queries, an on-demand approach of querying could be implemented, where critical variables are recovered only when needed. In addition, the prompt engineering could be improved such that each query contains information of multiple execution steps. To reduce the cost of each query, the future work could attempt to train a local deep learning model and query from this model instead of ChatGPT.

Furthermore, this project focuses on increasing the number of usable programs in Defects4J. The future work should study the effects of recovering data dependencies on reducing the length of path from the fault revealing step to the root cause during a debugging session. Due to the high cost of ChatGPT API, only 114 programs were tested. After the future work reduces the cost of the approach, it should be tested on an expanded dataset. Mutation on the existing programs could be performed to control and test data dependency recovery on a larger set of data structures.

3 Contributions to DebugPilot

3.1 Inputs and Outputs Detection

3.1.1 Definitions

Input: an input is a variable that is guaranteed to be correct based on the user's knowledge.
Output: an output is an incorrect variable during the execution.

3.1.2 Problem Statement

At the beginning of a debugging session with DebugPilot, the user is required to select the outputs. In an earlier version of DebugPilot, selection of the inputs was also required. As a result, there was a need to develop automatic detection of inputs and outputs (IO) to model user behaviour during the simulation experiments.

To control the level of familiarity on different programs, it was assumed that a user has complete knowledge about the program that he is debugging. Thus, both a buggy program and a program with the bug fixed were executed, with the fixed program representing the user's knowledge. By comparing the buggy and correct execution traces, the first discrepancy between the traces from the back was taken as the node containing the output. The output, either a variable or a conditional branch, was then obtained from this node. After this, the correct variables along the trace were recorded as the inputs.

This implementation has the limitation that the IO detected were sometimes inaccurate or threw exceptions. Thus, the existing algorithm for IO detection and the failing scenarios should be studied. The algorithm should then be improved to reduce the exceptions thrown.

Furthermore, IO Detection was repeated each time an experiment was conducted. However, the results of IO Detection on deterministic buggy programs from the dataset were fixed, making this repetition unnecessary. Therefore, support for storing and parsing the detected IO helps to save time and computational resources during experiments. In addition, storing the IO to a file makes it possible to manually correct detected IO that is inaccurate.

3.1.3 Storage and Correction of Detected Inputs and Outputs

To address the above-mentioned problems, a solution that involves storing the detected IO and providing support for manual correction was proposed. The architecture of the enhanced IO Processor is shown in Figure 1.

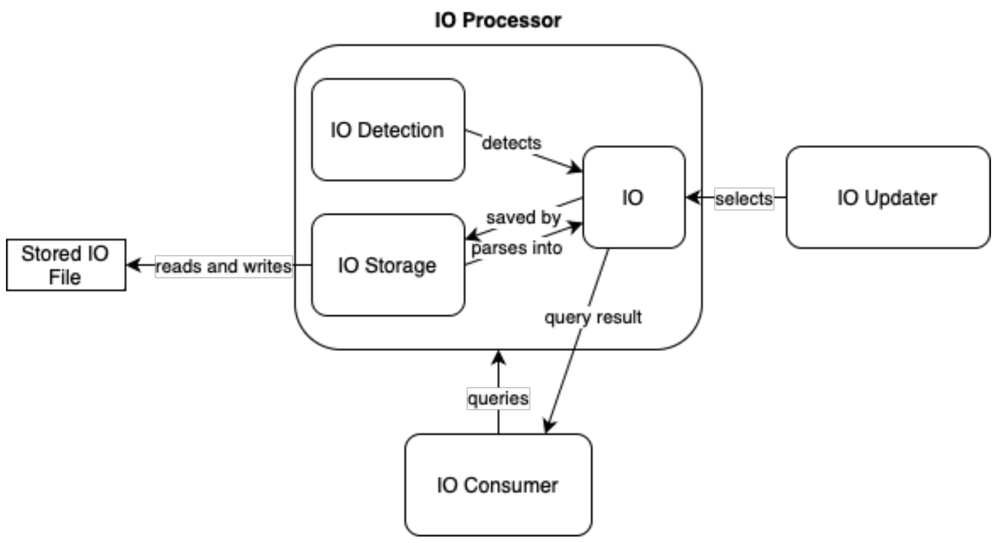


Figure 8. IO Processor Architecture

The IO Processor supports accessing IO obtained by either reading from the IO file or detecting from the buggy execution trace. In addition, it provides an interface for manual correction of the stored IO that are deemed to be inaccurate.

As shown in Figure 8, the IO Processor has three components: IO, IO Detection, and IO Storage. They correspond to the InputsAndOutput, IODetector, and StoredIOProcessor classes, and have the relationship as shown in Figure 9. This design decision of separating IODetector and StoredIOProcessor classes follows the Single Responsibility Principle, which makes the IO Processor easily maintainable and extensible.

In addition, the IOSelectionHandler class is the interface for manual correction of the inputs and outputs, and it acts as an IO Updater in Figure 8. It also provides an option to select the outputs only, in this case, the inputs will be detected by the IODetector based on the selected outputs. The selected IO are passed to StoredIOProcessor, which updates the existing IO file. On the other hand, the ProjectsDebugRunner class is responsible for running the automated experiments, that is, it acts as an IO Consumer. It calls StoredIOProcessor class to retrieve the stored IO. If such information doesn't exist, it calls IODetector to detect IO and stores the detected IO by StoredIOProcessor. The IOSelectionHandler practices Abstraction and Encapsulation by providing an interface to IO correction. In this way, the program becomes more flexible while protecting the IO file from unexpected modification.

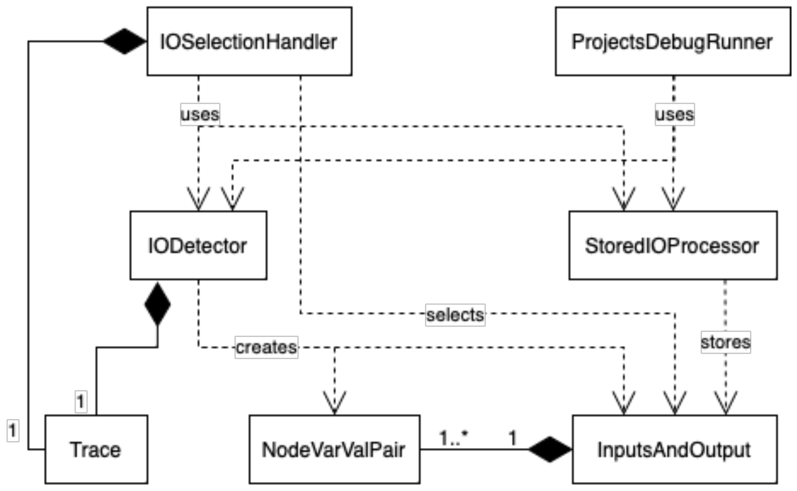


Figure 9. IO Processor Class Diagram

3.1.4 Updated Algorithm for Output Detection

To detect the output, the existing algorithm takes the first node from the back of the buggy trace that is different from the correct trace. This works when the output is a variable. However, when the error is caused by execution of a wrong control branch, the fault-revealing step should be the first condition result leading to this execution instead of the last executed step. To correct this, the algorithm of output detection was updated such that once a wrong control flow is detected, search the trace through control dominance relations until the first problem-causing condition is found. This condition is then taken as the wrong output of the buggy program. Such modification could be made because of the assumption that the user has complete knowledge about the code and knows which condition leads to the wrong control flow.

The updated algorithm for output detection is as follows:

Given a buggy trace T_{buggy} and a correct trace $T_{correct}$.

```
1. Start from the last node in $T_{buggy}$.
2. Traverse through the trace, compare each node $N_{buggy}$ on $T_{buggy}$ with the
   corresponding node $N_{correct}$ on $T_{correct}$ until one of the conditions is met:
   2a. If $N_{correct}$ cannot be found:
       // $N_{buggy}$ is on a wrong control flow.
       Search for and return the first wrong condition result that leads to this flow.
   2b. If $N_{correct}$ is found on $T_{correct}$, and $N_{buggy}$ contains wrong variables:
       Return these wrong variables as outputs.
```

3.1.5 Evaluation

The original algorithm threw IO Detection Exception when there are preceding steps between the conditional step and the last executed step. Since this conditional branch should not be executed, these preceding steps cannot be found on the correct trace. IO Detection Exceptions were thrown because the input detection compares the traces from the output node, but some nodes from the buggy trace cannot be found on the correct trace.

To evaluate the effectiveness of the enhancements described in Section 3.1.3 and 3.1.4, IO Detection was performed on 416 buggy programs from Defects4J (Just et al., 2014). Before these changes were made, DebugPilot threw IO Detection Exception on 89 programs, indicating failures in detection of inputs or outputs. The enhancements reduced this number to 9, significantly increased the number of usable bugs for the simulated experiments.

3.2 Formulation of Suspiciousness Function

3.2.1 Definitions

The definitions of Read and Written Variables are included in Section 2.1.2.

3.2.2 Problem Statement

In DebugPilot, the concept of suspiciousness was introduced to differentiate between possible paths originating from the output node. To approximate the suspiciousness of a written variable, the following equation was formulated:

$$sus(var_w) = sus(s) + sus(s, s_c) + \sum_{var_r \in V_r} sus(var_r)$$

where s is a writing step with control dominator s_c, V_r is the set of read variables, each read variable represented by var_r, and var_w is the written variable. The details of computation of $sus(s)$ and $sus(s, s_c)$ are omitted here, since they are irrelevant to the work in the report.

The last component is the combined suspiciousness score of read variables, which is a summation of the individual scores. The major limitation in this formula is that the combined score increases dramatically with increasing number of read variables. As a result, the combined score cannot reflect the overall suspiciousness accurately. For example, consider n variables with low and equal suspiciousness. Denote this suspicious score by s_l, and denote the combined score as $s_1 = \sum_{var_r \in V_r} sus(var_r)$. Then the combined score is $s_1 = n \cdot s_l$. If n is large enough, s_1 will be high even though all the variables have low suspiciousness, increasing the risk of classifying a correct step as suspicious.

A potential solution of this problem is to use the average score instead, that is, $s_2 = \frac{1}{|V_r|} \sum_{var_r \in V_r} sus(var_r)$. However, this solution has a flaw that variables with different levels of suspiciousness have equal weightage. Consider the case where there is one variable with high suspicious score and n variables with negligible and equal suspicious scores. Denote the high and low scores by s_H and s_L. Then the combined score is $s_2 = \frac{1}{n+1}(s_H + n \cdot s_L)$, and $\lim_{s_L \to 0} s_2 = \frac{s_H}{n+1}$. The combined score should be high due to the presence of a highly suspicious variable. However, in this example, s_2 evaluates to $\frac{s_H}{n+1}$ as $s_L \to 0$. The combined score decreases with increasing n, eventually leading to classifying a suspicious step as correct.

Since the existing equations suffer from high false positive or false negative rates, it is necessary to propose a more reasonable approximation of the combined suspicious score.

3.2.3 Solution

Instead of the above-mentioned solutions, the following formula was proposed and adopted:

$$sus(var_w) = sus(s) + sus(s, s_c) + \sqrt{\sum_{var_r \in V_r} sus(var_r)^2}$$

where the summation was replaced by $s_3 = \sqrt{\sum_{var_r \in V_r} sus(var_r)^2}$.

The inner term $\sum_{var_r \in V_r} sus(var_r)^2$ can be considered as a weighted sum of the suspicious scores, such that a more suspicious variable contributes more to the overall suspiciousness. This addresses the problem with the unweighted average s_2 as described in Section 3.2.2. The combined score is $s_3 = \sqrt{s_H^2 + n \cdot s_L^2}$ based on the updated equation. Consequently, $\lim_{s_L \to 0} s_3 = \sqrt{s_H^2 + 0} = s_H$, which means the highly suspicious variable dominates the combined score as expected.

Furthermore, by taking the square root of the weighted sum, the problem with s_1 is prevented. Consider the example for s_1 described in Section 3.2.2, the combined score is updated to be $s_3 = \sqrt{n \cdot s_l^2} = \sqrt{n} \cdot s_l$, which increases much slower with increasing n compared to the original solution.

In addition, the proposed solution has a comprehensive geometric interpretation. Treat the suspicious scores of the read variables as orthogonal vectors, then the combined score represents the magnitude of the sum of these vectors. Figure 10 illustrates this geometric interpretation with $|V_r| = 3$.

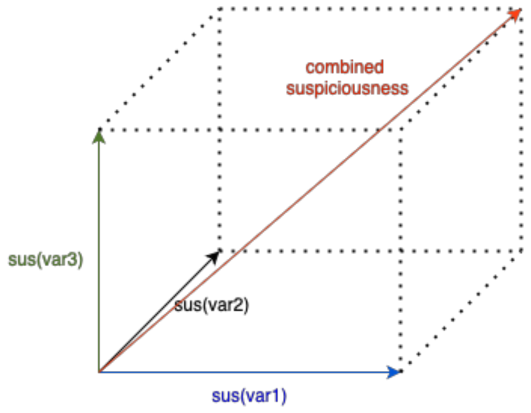


Figure 10. Geometric Interpretation of the Combined Suspiciousness of 3 variables

4 Conclusions

Data Dependency Recovery

The work has identified the need of recovering missing data dependencies in software engineering research. Based on the current trace generation of Microbat, this work has identified two sources for the problem of missing data dependencies, namely incomplete instrumentation and partial recording of variable values. To address this problem, three existing solutions and their limitations have been studied. After this, a solution leveraging LLMs with an extensible framework has been proposed. The overall workflow involves conversion of execution information to LLM prompt, invocation of LLM APIs, and conversion of natural language representations of variables to variables on an execution trace. Under this framework, a benchmark with hard coded LLM responses and two versions of the solution have been implemented and evaluated. By comparing the results, the effectiveness of the proposed solution on addressing the two sources of missing data dependencies has been demonstrated.

Based on the analysis of runtime overhead of the proposed solution, a potential future research direction of reducing the cost of LLM queries used in this solution has been pointed out. From the perspectives of reducing the number of queries and reducing the cost of each query, potential future works involve on-demand querying and utilizing local deep learning models. In addition, this work has pointed out the need of scaling up the experiment through test case mutation after cost is reduced.

Contributions to DebugPilot

In this work, DebugPilot has been enhanced and tested in different ways. First, the IO Detection for conducting simulation experiments was improved. The updated algorithm for output detection reduced the number of programs throwing IO Exceptions, significantly increased the number of usable bugs for simulation experiments. The storage of automatically detected IO not only save time and computational resources during the simulation experiments, but also enabled manual correction of inaccurate detection.

Apart from efforts on improving the simulation experiments, the computation of the combined suspicious score of the read variables has been improved. This enhancement addressed the issues of dramatic increase of the score with increasing number of variables and variables with different levels of suspiciousness having the same contribution to the overall score. The proposed solution also has a geometric interpretation that demonstrates the reasonableness of this combined score.

References

Agrawal, H., DeMillo, R. A., & Spafford, E. H. (1993). Debugging with dynamic slicing and backtracking. *Software: Practice and Experience*, *23*(6), 589-616.

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers : Principles, techniques, and tools* (2nd ed., pp. 597-617). Pearson Education, Inc.

Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., & Zhang, J. M. (2023). Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533*.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., ... & Zhou, M. (2020). Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Just, R., Jalali, D., & Ernst, M. D. (2014, July). Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis* (pp. 437-440).

Lin, Y., Sun, J., Xue, Y., Liu, Y., & Dong, J. (2017, May). Feedback-based debugging. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (pp. 393-403). IEEE.

Ma, W., Liu, S., Wang, W., Hu, Q., Liu, Y., Zhang, C., ... & Liu, Y. (2023). The scope of chatgpt in software engineering: A thorough investigation. *arXiv preprint arXiv:2305.12138*.

OpenAI. (2022, Nov 30). Introducing ChatGPT. *OpenAI*. https://openai.com/blog/chatgpt

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., ... & Ma, S. (2020). Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Wang, H., Lin, Y., Yang, Z., Sun, J., Liu, Y., Dong, J., Zheng, Q., & Liu, T. (2019). Explaining regressions via alignment slicing and mending. *IEEE Transactions on Software Engineering*, *47*(11), 2421-2437.

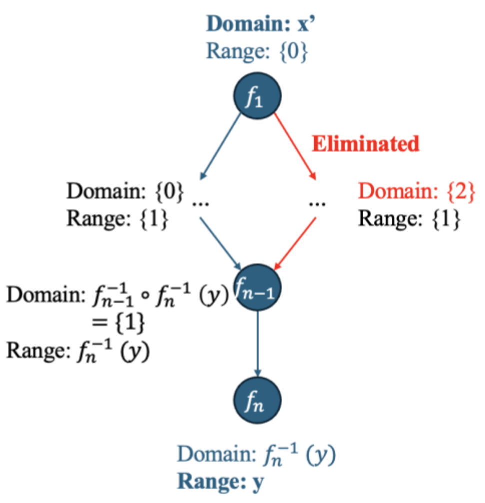
Wong, Y., Lin, Y., Wang, H., Pei, Y., Wang, J., Dong, J., & Mei, H. (2023). *DebugPilot: Automate Debugging via Synthesizing Interactive Debugging Progresses.* [Unpublished Manuscript].

Wu, S. (2024, March 12). Introducing Devin, the first AI software engineer. *Cognition Labs*. https://www.cognition-labs.com/introducing-devin

Xu, Z., Ma, S., Zhang, X., Zhu, S., & Xu, B. (2018, May). Debugging with intelligence via probabilistic inference. In *Proceedings of the 40th International Conference on Software Engineering* (pp. 1171-1181).

Yang, J., Jimenez C. E., Wettig A., Yao S., Narasimhan K., Press O. (2024, April). SWE-agent: Agent Computer Interfaces Enable Software Engineering Language Models. *SWE-agent*. https://swe-agent.com

Appendix A



A data flow can be represented as a series of functions $f_1, f_2, \ldots, f_n$

$$x = f_1^{-1} \circ \ldots \circ f_{n-1}^{-1} \circ f_n^{-1}(y)$$

Algorithm:

1. Represent each node as an inversible function f
2. Given the output y (execution result), find the input domain x
3. If x is different from ground truth x', eliminate this data flow

Appendix B

charAt method source code:

```
     public char charAt(int index) {
        if (isLatin1()) {
            return StringLatin1.charAt(value, index);
        } else {
            return StringUTF16.charAt(value, index);
        }
    }
```

Decompiled charAt method after enhanced instrumentation:

```
   public char charAt(int arg0) {
      boolean $shouldExecute =
ExecutionTracer._shouldExecuteInjectedCode();
      if (!$shouldExecute) {
         if (arg0 >= 0 && arg0 < this.value.length) {
            return this.value[arg0];
         } else {
            throw new StringIndexOutOfBoundsException(arg0);
         }
      } else {
         String $className = "java.lang.String";
         String $methodSignature = "java.lang.String#charAt(I)C";
         Object[] $tempVar3 = new Object[]{arg0};
         IExecutionTracer $tracer = ExecutionTracer._getTracer(false,
$className, $methodSignature, 657, 660, "arg0", "I", $tempVar3);
         $tracer._hitLine(657, $className, $methodSignature, 1, 0,
"iload_1[27](1):iflt[155](3) ->
13:iload_1[27](1):aload_0[42](1):getfield[180](3)
3:arraylength[190](1):if_icmplt[161](3) -> 22:");
         if (arg0 >= 0) {
            char[] var10003 = this.value;
            $tracer._readField(this, var10003, "value", "char[]", 657,
$className, $methodSignature);
            if (arg0 < var10003.length) {
               $tracer._hitLine(660, $className, $methodSignature, 2, 0,
"aload_0[42](1):getfield[180](3)
3:iload_1[27](1):caload[52](1):ireturn[172](1):");
               char[] var10002 = this.value;
               $tracer._readField(this, var10002, "value", "char[]", 660,
$className, $methodSignature);
               char $tempVar2 = var10002[arg0];
               $tracer._readArrayElementVar(var10002, arg0, $tempVar2,
"char", 660, $className, $methodSignature);
               $tracer._hitReturn(Integer.valueOf($tempVar2), "I", 660,
$className, $methodSignature);
               $tracer._hitMethodEnd(660, $className, $methodSignature);
               return $tempVar2;
            }
         }

         $tracer._hitLine(658, $className, $methodSignature, 0, 0,
"new[187](3) 6:dup[89](1):iload_1[27](1):invokespecial[183](3)
7:athrow[191](1):");
         StringIndexOutOfBoundsException var10000 = new
StringIndexOutOfBoundsException;
         Object[] $tempVar1 = new Object[]{arg0};
         $tracer._hitInvoke(var10000,
"java.lang.StringIndexOutOfBoundsException",
"java.lang.StringIndexOutOfBoundsException#<init>(I)V", $tempVar1, "I",
"V", 658, $className, $methodSignature);
         var10000.<init>((Integer)$tempVar1[0]);
         $tracer._hitMethodEnd(658, $className, $methodSignature);
         throw var10000;
      }
   }
```

Appendix C

Prompt Version 1

“Return in the following format: <method type><method action><name of internal array><index>.
<method type> can take values from <get/set>, it represents whether the line of code gets or sets values in the invoking object. When a method is <get>, the tags afterward are not needed.
<method action> can take values from <add/remove/replace>, it represents the type of operation done by the line of code. When a method is <remove>, the tags afterward are not needed.
<name of internal array> represents the name of the internal array used to store elements in a data structure.
<index> can take values from <start/end/all/index/key>, they represent the range of positions being modified in the internal array that stores the elements.
For example:
"list.add(object)" with signature "java.util.List#add(Ljava/lang/Object;)V":<set><add><elementData><end>,
"list.add(int, object)" with signature
"java.util.List#add(ILjava/lang/Object;)V":<set><add><elementData><index>,
"list.set(int, object)" with signature
"java.util.List#set(ILjava/lang/Object;)Ljava/lang/Object;":<set><replace><elementData><index>,
"list.get(int)" with signature "java.util.List#get(I)Ljava/lang/Object;":<get>,
"list.remove(int)" with signature
"java.util.List#remove(I)Ljava/lang/Object;":<set><remove><elementData><index>,
"hashmap.put(int, object)" with signature
"java.util.HashMap#put(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;":<set><add><table><key>,
"hashmap.forEach(biConsumer)" with signature
"java.util.HashMap#forEach(Ljava/util/function/BiConsumer;)V":<set><replace><table><all>,
"hashset.add(object)" with signature "java.util.HashSet#add(Ljava/lang/Object;)Z":<set><add><table><key>.
Then”

Prompt Version 2

“Let {name:var_name,type:var_type,value:var_value} represent a variable. Return the fields in var_value that are modified. In your response, do not explain and return strictly in this format:<field1_var_name>;<field2_var_name>;...;<fieldn_var_name>
e.g., Given variable
{name:l1,type:java.util.ArrayList,value:[{name:elementData,type:java.lang.Object[],value:[{name:com/mycompany/app/AppTest{103,151}l1-1.elementData[0],type:java.lang.Integer,value:1};]};{name:size,type:int,value:1};]} After calling "l1.add(0,3)" once, the following fields of "l1" are modified:<l1#elementData#com/mycompany/app/AppTest{103,151}l1-1.elementData[0]>;<l1#elementData#com/mycompany/app/AppTest{103,151}l1-1.elementData[1]>;<l1#size>
Then”