2017

CONTEXT-AWARE DEBUGGING FOR CONCURRENT PROGRAMS

Justin Chu
University of Kentucky, justinchu@uky.edu

Digital Object Identifier: https://doi.org/10.13023/ETD.2017.505

Justin Chu, Student

Dr. Tingting Yu, Major Professor

Dr. Miroslaw Truszczynski, Director of Graduate Studies

CONTEXT-AWARE DEBUGGING FOR CONCURRENT PROGRAMS

THESIS

A thesis submitted in partial
fulfillment of the requirements for
the degree of Master of Science in
the College of Engineering at the
University of Kentucky

By
Justin Chu
Lexington, Kentucky

Co-Directors: Dr. Tingting Yu, Assistant Professor of Computer Science
and Dr. Jane Hayes, Professor of Computer Science
Lexington, Kentucky

2017

ABSTRACT OF THESIS

Context-Aware Debugging for Concurrent Programs

Concurrency faults are difficult to reproduce and localize because they usually occur under specific inputs and thread interleavings. Most existing fault localization techniques focus on sequential programs, but fail to identify faulty memory access patterns across threads, which are usually the root causes of concurrency faults. Moreover, existing techniques for sequential programs cannot be adapted to identify faulty paths in concurrent programs. While concurrency fault localization techniques have been proposed to analyze passing and failing executions and identify faulty access patterns, they primarily focus on using statistical analysis. We present a novel approach to fault localization using feature selection techniques from machine learning. Our insight is that the concurrency access patterns obtained from a large volume of coverage data generally constitute high dimensional data sets, yet existing statistical analysis techniques for fault localization are usually applied to low dimensional data sets. Each additional failing or passing run can provide more diverse information, which can help localize faulty concurrency access patterns in code. The patterns with maximum feature diversity information can point to the most suspicious pattern. We then apply data mining technique and identify the interleaving patterns that are occurred most frequently and provide the possible faulty paths. We also evaluate the effectiveness of fault localization using test suites generated from different test adequacy criteria. We have evaluated Cadeco on 10 real-world multi-threaded Java applications. Results indicate that Cadeco outperforms state-of-the-art approaches for localizing concurrency faults.

KEYWORDS: Concurrent Program, Fault Localization, Debugging, Machine Learning, Data Mining, Empirical Study

Author's signature: Justin Chu

Date: December 15, 2017

Context-Aware Debugging for Concurrent Programs

By
Justin Chu

Co-Director of Thesis: Dr. Tingting Yu

Co-Director of Thesis: Dr. Jane Hayes

Director of Graduate Studies: Dr. Miroslaw Truszczynski

Date: December 15, 2017

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1 Introduction

The advent of multicore processors has greatly increased the prevalence of concurrent software. Not surprisingly, failures due to concurrency faults occur prolifically in deployed concurrent systems. Debugging software after it has failed is an expensive part of the software-development process. Finding the fault that has caused the failure in concurrent programs is very challenging because a concurrency fault typically involves a particular interleaving of multiple threads; this reduces understandability. In addition, when a concurrency fault is exposed, it is often difficult to reproduce the fault due to the nondeterminism of concurrent programs.

Over the past decades, there has been a great deal of work on detecting particular types of concurrency faults based on access patterns, such as data races [6,12,32,42], atomicity violations [4,13,52,60,66], and deadlock [3,31,55]. However, while these techniques may report many patterns, only some of which might directly identify the fault. In addition, these methods do not presently have any way to rank or prioritize the patterns.

Spectrum-based fault localization techniques [2,9,30,51,67] have been used to automatically identify and rank likely faulty locations. Wong et al. provide a survey [64]. These techniques use dynamic information obtained from failing and passing test executions (called spectra) to compute suspiciousness scores for each coverage target. The underlying intuition of these methods is that similar test cases generate similar execution spectra while dissimilar test cases generate very different types of execution spectra. The runtime information, e.g., statements executed and branches taken, of a program is collected with respect to a set of positive test cases (termed passing runs) and negative test cases (termed failing runs). The premise is that data analysis of two different types of spectra may pinpoint the location of the bug.

However, most existing spectrum-based fault localization techniques focus on sequential programs that rank program elements such as statements. Falcon [47] and CCI [29] are the most closely related work on applying spectrum-based fault localization techniques to concurrent programs. Falcon combines pattern identification with statistical rankings of suspiciousness of those patterns. CCI leverages statistical debugging and views interleavings as predicates which are collected in the runtime and analyzed to find the location of the concurrency faults. While these techniques have shown promise, effectively and efficiently locating faults still remains a challenging problem. In our experiment (Section 3.4.4), Falcon yields inaccurate results in 30% of its subject programs.

While precisely localizing concurrency bugs in high dimensional data is challenging, additional challenges arise as existing techniques do not provide sufficient contextual information for debugging. The contextual information, such as inter-thread control and data dependency, is often useful for understanding how highly ranked patterns lead to program failures. Existing research [9,28] has shown that execution paths can help isolate bugs more precisely by providing more information about the context in which bugs occur. For example, Jiang et al. [28] propose techniques to

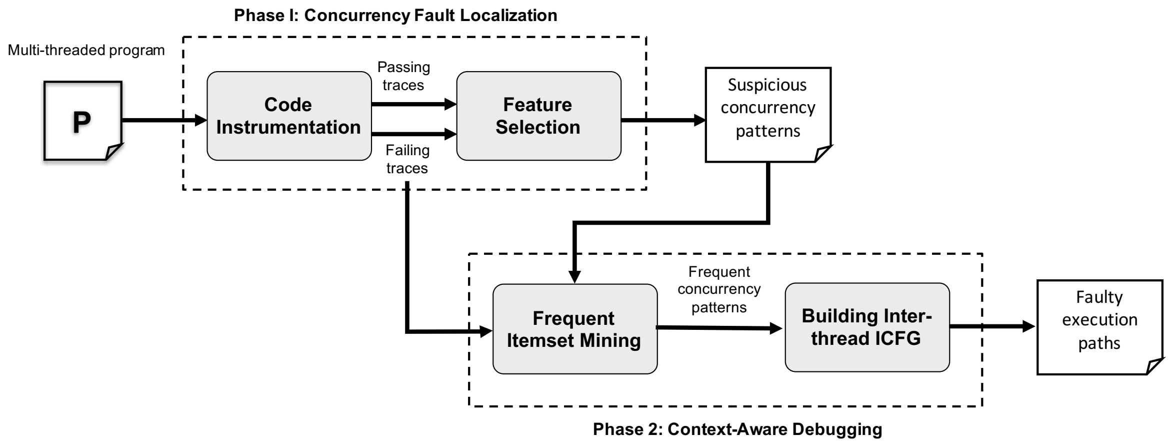

Phase I: Concurrency Fault Localization
Multi-threaded program
P
Code Instrumentation
Passing traces
Failing traces
Feature Selection
Suspicious concurrency patterns
Frequent Itemset Mining
Frequent concurrency patterns
Building Inter-thread ICFG
Faulty execution paths
Phase 2: Context-Aware Debugging

Figure 1.1: The overview of our Cadeco framework

first identify buggy predicates, and then predict correlations and control flow paths that connect the buggy predicates and correlated predicts for better understanding the bugs. However, these techniques focus on sequential software and have not considered concurrent software systems.

Given the foregoing discussion, the overall goal of our research is to provide an automated **c**ontext-**a**ware **de**bugging framework for **co**ncurrent programs, Cadeco, that can 1) report a ranked list of interleaving patterns that are likely to induce the concurrency failure, and 2) provide intra- and inter-thread control-flow paths that connect the interleaving patterns to help developers better diagnose the concurrency failure.

Figure 1.1 provides an overview of Cadeco. First, it executes existing test inputs to obtain both passing and failing executions. During the execution, we randomly permute thread interleavings in order to generate execution traces with diverse patterns. Next, it leverages feature selection techniques used in machine learning to identify interleaving patterns that are likely to cause the failure. In the third step, Cadeco applies frequent pattern pattern mining technique on the failing execution traces to identify interleaving patterns that are frequently executed. Finally, Cadeco traverses the program's inter-thread inter-procedural control flow graph (IICFG) to identify faulty control flow paths that connect the faulty interleaving patterns and frequent executed patterns.

Cadeco differs from existing spectrum-based concurrency fault localization tools [29, 47] in several aspects. First, existing techniques obtain traces from random interleavings, but the quality of interleavings must be considered to ensure the diversity of covered patterns among failing and passing executions. We leverage the inter-thread def-use coverage criterion [38] for creating thread interleavings. We also investigate the effects of using an interleaving reduction strategy on the effectiveness of fault localization techniques. Second, standard feature selection techniques often identify features that can differentiate instances of different classes, but do not differentiate features that characterize one class from those that characterize another. Cadeco employs a constrained feature selection technique [35] that associates the patterns to particular failing or passing cases.

Cadeco also differs from existing context-aware debugging techniques [9, 28, 45]. First, existing techniques for sequential programs cannot be adapted to identify faulty paths in concurrent programs, because they consider the control and data flow across multiple threads. Second, Griffin [45] is the only work that considers using contextual information to improve comprehension of concurrency bugs. Griffin clusters the faulty concurrency patterns and outputs the grouped patterns along with suspicious methods. However, like the techniques for sequential programs, Griffin reports a list of methods around the faulty pattern but does not provide inter-thread level execution context.

Cadeco has been implemented as a software tool using Soot instrumentation [56], Weka machine learning tool [19] and SPMF [57]. To evaluate Cadeco, we conducted two experiments on 10 popular Java applications and known concurrency failures. The goal of our first experiment is to evaluate the feature selection techniques on concurrency bug localization. Our experimental results show that Cadeco can accurately rank faulty concurrency patterns in the top-2 for nine out of 10 applications. When compared to Falcon, Cadeco has the same level of accuracy as Falcon and is more accurate for two out of 10 subjects. Our results also show that execution trace reduction causes negligible impact on the effectiveness of the fault-localization technique. The second experiment evaluates the effectiveness and efficiency of our context-aware debugging technique. The results show that the faulty control flow paths reported by Cadeco are accurate and useful for understanding concurrency failures.

In summary, this thesis makes the following contributions:

- We develop a spectrum-based fault localization technique for concurrent programs using feature selection techniques.
- We develop a context-aware debugging technique for concurrent programs using a novel combination of feature selection, data mining, and control flow graph analysis.
- We implement out techniques in a tool, Cadeco, and conduct empirical studies to demonstrate Cadeco's effectiveness and efficiency in real-world applications.

In the next chapter, we present background and use examples to illustrate the main technical challenges and our corresponding solutions. We then describe our techniques on concurrency fault localization in Chapter 3. Next, we present our detailed algorithms on context-aware debugging in Chapter 4. Finally, we give our conclusions and future work in Chapter 5.

Chapter 2 Background and Motivation

2.1 Background

In this section, we provide background on spectrum-based fault localization and feature selection. We then show two motivating examples.

2.1.1 Spectrum-based Fault Localization Techniques

Spectrum-based Fault Localization (SBFL) is one of the most popular approaches for locating software faults because of its simplicity and effectiveness [65]. SBFL utilizes the execution result (passing or failing) of each test case and the corresponding coverage information to evaluate the likelihood of each program entity (e.g., statements, basic blocks, methods, files) being faulty. A SBFL technique returns a ranked list of program entities sorted by their suspiciousness scores. Developers can then manually inspect program elements, starting from the most suspicious element until the root cause of the fault is identified.

Different formulas for computing the suspiciousness scores have been proposed based on heuristics [30, 67]. A well-known statistics-based formula is used by Tarantula [30]. Tarantula computes the suspiciousness score of a program element e as follows:

$$Suspicious(e) = \frac{\frac{n_{f(e)}}{n_f}}{\frac{n_{f(e)}}{n_f} + \frac{n_{p(e)}}{n_p}}$$

Here, $n_{f(e)}$ is the number of failing executions that cover a program element e and $n_{p(e)}$ is the number of passing executions that cover e.

2.1.2 SBFL for Concurrent Programs

The most closely related approach that uses spectrum-based fault localization for concurrent programs is Falcon [47]. Falcon first collects shared memory access patterns between threads during program executions and then outputs a set of patterns ranked by suspiciousness. Falcon views each access pattern as a program entity e and uses the Tarantula formula to compute the suspiciousness scores.

Our works studies an alternative approach to localizing concurrency faults. Since concurrency access patterns in the coverage generally involve high dimensional data, we hypothesize that the feature selection methods in machine learning may yield more accurate results compared to the statistical analysis approach. For example, the data used in the Tarantula formula involves only three dimensions: patterns, failing executions, and passing executions.

There has been some research on using feature selection for fault localization [35, 49]. For example, Roychowdhury et al. [49] applied feature selection methods to localize faults in seven programs and the results were more precise than statistical analysis methods. However, existing techniques focus on sequential programs and have not studied the cost-effectiveness of fault localization in concurrent programs.

2.1.3 Feature Selection Techniques

Machine learning algorithms often deal with data sets that are characterized in high-dimensional feature space. High-dimensional feature space may cause certain learning algorithms to perform poorly [11] due to the exponential increase in volume associated with adding extra dimensions, a phenomenon known as the "curse of dimensionality".

Therefore, machine learning algorithms are often evaluated in reduced feature spaces. Toward that end, an important technique is to identify useful features and discard irrelevant and redundant ones; this process is called feature selection [5].

Feature selection is often classified into three categories: filter, wrapper, and embedded. Filter feature selection algorithms estimate a score for each feature based on a statistical measure. Features with scores greater than a predefined threshold are included in the output of a filter feature selection algorithm. In our study, we consider the following algorithms [63]: Chi-Square Attribute evaluation (CH), Gain-Ratio Attribute evaluation (GR), Information-Gain Attribute evaluation (IG), Relief Attribute evaluation (RF), and Symmetrical Uncertainty Attribute evaluation (SU). The wrapper algorithm selects a subset of features using a learning algorithm, which is somewhat similar to the model selection process. The wrappers usually perform better as the target learning algorithm is used to select the features. However, it becomes computationally expensive when the dimensions of the data sets are large. The embedded algorithm selects features during the learning processes, typically in an iterative fashion. Decision tree is a classic example of this approach.

2.1.4 Concurrency Bug Patterns

Cadeco considers three types of concurrency bugs – order violations, atomicity violations, and deadlock – three of the most important types of concurrency bugs [39]. An order violation occurs when a pair of memory accesses between two threads, in which one of the memory accesses is a write, leads to unintended program behavior. An atomicity violation occurs when a set of accesses in an atomic region is interfered with by other accesses in a different thread, and that interference leads to unintended program behavior. A deadlock occurs when one or more threads block forever waiting for a synchronization event that will never happen. There has been research on identifying concurrency violation patterns [41, 46, 54]. The common patterns are summarized in Table 2.1. The first three patterns involve order violations, the 4^{th} to the 8^{th} patterns are atomicity violations, and the remaining patterns are deadlocks.

To illustrate, Figure 2.1 shows snippets from the ArrayList class of the Java Collection Library. An atomicity violation occurs when the program executes in order 2→3→13→14→4→7→8. In lines 2–3, the program retrieves size of `c` and initializes an array of Object with the size. However, in lines 13–14, the array and size are increased by the `addAll` method. Then, in lines 4–9, the increased array is

Table 2.1: Memory-access patterns

ID	*Pattern*	*Description*
1	(R_1, W_2)	Unexpected value is written
2	(W_1, R_2)	Unexpected value is read
3	(W_1, W_2)	The result of remote write is lost
4	(R_1, W_2, R_1)	Modifies local read
5	(W_1, W_2, R_1)	Modifies local read
6	(W_1, R_2, W_1)	Modifies local write
7	(R_1, W_2, W_1)	The result of remote write is lost
8	(W_1, W_2, W_1)	The result of remote write is lost
9	(wait, wait)	deadlock
10	(wait, notify)	deadlock
11	(wait, notifyAll)	deadlock

Thread 1

```
1.  public ArrayList(Collection c) {
2.    size = c.size(); // write
3.    array = new Object[size]; // write
4.    c.toArray(array);
5.  }

6.  public Object[] toArray(Object a[]) {
7.    copyarray(array, 0, a, 0, size); // read
8.    return a;
9. }
```

Thread 2

```
10. boolean addAll(Collection c) {
11.   Object[] a = c.toArray();
12.   int numNew = a.length;
13.   copyarray(a, 0, array, size, numNew); // write
14.   size += numNew; // read, write
15.   return numNew!=0;
16. }
```

Figure 2.1: An real concurrency bug from the ArrayList class of the Java Collection Library

Table 2.2: Patterns associated with the scores computed by a feature selection method

Pattern	E1	E2	E3	E4	Score
size: (W^2, W^{14}, R^7)			$*$	$*$	1.0
size: (R^7, W^{14})	$*$	$*$	$*$		0.33
array: (R^7, W^{13})	$*$	$*$	$*$		0.33
array: (W^{13}, R^7)		$*$	$*$	$*$	0.66
Result	P	P	F	F	

copied to array of Object a, and thus the program may crash with an exception. In this example, the bug involves a write-write-read ($W_1W_2R_1$) pattern (i.e., the write access is from Thread 1, the write access is from Thread 2, and the read access is from Thread 1), and the access order is 2→14→7. The second pattern involves array: write-read (W_2R_1) pattern (i.e., the write is from Thread 2, and the read access is from Thread 1), and the access order is 13→7.

Cadeco first employs the algorithm in Unicore [46] that can detect both order and atomicity violations using memory-access patterns. The technique monitors memory accesses with a fixed-sized window [47] and combines the accesses into patterns. In addition, Cadeco detects deadlock patterns by monitoring wait – notify synchronization operations, including wait, notify, and notifyAll.

Table 2.2 shows the results of four executions and the ranking of identified concurrency patterns. The first column shows all concurrency violation patterns identified from the four executions. Columns 2–5 list all four executions. If a pattern appears in an execution, the cell is marked with $*$. For example, (W^{2}, W^{14}, R^{7}) pattern for shared variable `size` appears in executions 3 and 4; the superscript indicates the line number. The last row indicates the result of each execution (passing or failing). The last column reports the feature selection score for each pattern after applying a feature selection technique. As the table shows, the first pattern has the highest score followed by the fourth pattern. The two patterns are the root causes of the concurrency bug.

Existing concurrency bug detection techniques cannot precisely identify the root cause of failures. These techniques usually report all or some of the four distinct patterns shown in the first column of Table 2.2 after the four executions. For instance, all patterns will be reported after the third execution because they all appear in the failing executions.

2.1.5 Frequent Itemset Mining

Frequent itemset mining is a sub-branch of data mining that focuses on looking at sequences of actions or events. It has been widely adopted in different areas. The basic idea of frequent itemset mining is to find all common sets of items. Those itemsets should exists at least a minimum amount of times. An example of a set of items would be grocery items bought in a trolley. The algorithm scans the trolley multiple times, and find the items that occur more frequency in each trolley than a given threshold. The number of occurrence is called support, and the threshold is called the minimum support.

We applied frequent itemset mining on the traces we collected from the fault localization phase. In our case, the interleaving patterns found by each run is like the grocery items we put in each trolley.

A frequent itemset is an itemset that appears in at least minimum support (minsup) items from the trolley. A frequent maximal itemset is a frequent itemset that is not included in a proper superset that is a frequent itemset.

There are a few different approaches in frequent itemset mining. We chose the FP Maximal (FP-Max) approach to mine the frequent patterns as it is more efficient than the others. FP-Max uses frequent-pattern tree to represent the frequent items. It applies the divide and conquer strategy by dividing the data into a set of conditional data, and mined each of them separately. This reduces the searching time.

One of the important factors in FP-Max is the support. The support is a value to demonstrate how frequently the itemset occurs in the dataset. The support of X with respect to I is defined as the proportion of items i in the dataset which contains the itemset X. Table 2.3 shows an example.

$$\mathit{SUP(X)}{=}\frac{|\{i \in I; X \subseteq i\}|}{|I|}$$

Table 2.3: Frequent Itemset Mining Suport Example

test id	pattern 1	pattern 2	pattern 3	pattern 4
1	0	0	0	0
2	1	1	0	0
3	0	0	1	0
4	0	0	0	1
5	0	0	1	0

Itemset X = { pattern 1, pattern 2 } has a support of 1/5 = 0.2. It appears in 1 out of 5 tests (20%).

2.2 Related Work

There has been a great deal of work on testing and debugging concurrent programs. In this section, we discuss the work most relevant to our own and contrast existing methods to the Cadeco approach.

2.2.1 Fault Localization

Automated Fault Localization Automatic bug localization or automatic debugging has been an active research area for over two decades [53]. Existing techniques can be broadly categorized into two categories: dynamic [1] and static [22]. Generally, dynamic fault localization techniques can localize bugs very precisely (such as at the statement level). However, they require a test case suite and execution of the program to gather passing and failing execution traces. Furthermore, the approaches are computationally expensive. Spectrum-based fault localizations [1, 30], dynamic slicing [71], delta debugging [68] are some of the well known techniques in this category. Our work focuses on spectrum-based fault localization.

Spectrum-Based Fault Localization Tarantula uses the coverage of statements in the execution traces of failed and passed tests to compute the suspiciousness of each statement [30]. The statements with the highest suspiciousness score must be examined first when looking for the fault. If the fault is not found, the remaining statements are examined in a non-increasing order of their suspiciousness scores. Other approaches such as Pinpoint, AMPLE, and Ochiaii [2] adopt the general framework of Tarantula but use different metrics to compute suspiciousness of statements. All of these techniques use statistical analysis.

Roychowdhury et al. [49] use a machine learning technique for localizing faults at the statement level. However, their technique does not study the effectiveness of feature selection when applied to concurrent programs.

Concurrency Fault Localization There has been much research on fault localization for concurrent programs [46,47,58,61,69]. For example, Wang et al. [61] identify shared memory access pairs that behave distinctively in failed and successful runs, and pinpoint root causes using different test procedures. This technique targets order violations and does not rank concurrency violation patterns. Park et al. [47] monitor memory-access patterns associated with a program's pass/fail results. CCI [29] ranks only shared variable accesses (predicates) and thus provides less contextual information. Both techniques then use statistical debugging to report data access patterns with suspiciousness scores. The main distinction of Cadeco is that it takes advantage of machine learning techniques for handling high dimensional data and studies the cost-effectiveness of using different methods to rank concurrency violation patterns.

Concurrency Fault Detection There has been much research on detecting concurrency violation patterns, involving data races [14,44], atomicity violations [40,70], and order violations [21,34]. However, as described in Chapter 1, these techniques target a specific type of bug. In addition, they may either report many false positives or do not rank patterns based on their suspiciousness.

2.2.2 Context-Aware Debugging

Context-Aware Debugging There has been some work on context-aware debugging [8,9,23,28]. Jiang et al. [28] use bug predictors and traverse the program's control flow graph to identify faulty control flow paths. However, they are not working on concurrent programs. Chilimbi et al. [9] apply statistical debugging and use path profiles to perform bug isolation. However, these techniques focus on sequential programs and do not track data or control flow across multiple threads. Cheng et al. [8] propose a discriminative graph mining algorithm for identifying bug signatures, which are on block and method level. Different from Cadeco, they use two sets of graphs as input from correct and faulty executions and search for a subgraph. In Cadeco, we use a full graph and frequent faulty patterns to search for possible paths. Griffin [45] is the most closely work to Cadeco – it first collects concurrency inter-thread read-write patterns and then generates a sequence of methods containing these patterns. However, the execution context output by Griffin is coarse-grained (i.e., method) and may not provide enough information. In addition, Griffin provides context only within individual threads and does not describe the execution flow across multiple threads.

Chapter 3 Concurrency Fault Localization Using Feature Selection

3.1 Introduction

In this chapter, we present a fault localization technique using feature selection for conncurrent programs. We hypothesize that the problem of fault localization based on data access patterns can be formulated into *feature selection* [18] used in machine learning, in which a subset of features is selected from all features to improve the prediction accuracy. In concurrency fault localization, the concurrency data access patterns can be formulated as features, so the problem can be viewed as selecting the access patterns that are likely to predict the failing executions. We conjecture that there are certain benefits of using a machine learning-based approach to localize concurrency because code coverage data is generally high dimensional data. A new failing/passing execution can provide additional information, where the data access pattern with maximum information diversity is most suspicious [49]. We evaluate the effectiveness and efficiency of our approach by comparing it to existing techniques.

3.2 Approach

Our technique for localizing concurrency faults consists of two major steps. In the first step, Cadeco monitors memory accesses and synchronization operations at runtime and detects concurrency violation patterns. The patterns are associated with passing and failing executions. Cadeco employs two strategies for gathering patterns. First, Cadeco explores different interleaving schedules for each input to maximize the diversity of patterns being covered. Since concurrent programs are non-deterministic, failures can be exposed in different executions under the same test input. To do this, Cadeco uses bytecode instrumentation to inject delay points before or after the shared variables. Second, Cadeco studies the effectiveness of a test case reduction technique in fault localization by removing traces containing the same concurrency violation patterns. Yu et al. [67] study the differences between the effectiveness of several fault-localization techniques on the unreduced and reduced test suites. However, their study does not target concurrent programs. Our evaluation shows that the impact of test case reduction on fault localization is negligible.

In the second step, Cadeco applies feature selection algorithms to identify patterns that have the most discriminative power in predicting the failing executions. Our intuition is that there exists a commonality between the concepts of feature selection and concurrency fault localization. Specifically, concurrency bug patterns can be viewed as features and the concurrency violation patterns covered in all executions are generally high dimensional data. The output is a list of suspicious patterns with their scores ranked from the highest to lowest.

3.2.1 Feature Selection

Our feature selection algorithm takes as input a set of passing and failing executions generated by exercising N test inputs together with their interleavings on the faulty program. Cadeco builds a coverage matrix, where each row indicates the identifier of an execution and each column indicates a unique concurrency bug pattern. The value of a matrix cell $X[i,\ j]$ is 1 if pattern j is covered in the i execution and 0 otherwise. An execution is labeled "P" if the corresponding execution is passed and "F" otherwise. The set of labeled executions are then given to a feature selection algorithm and the output is a ranked list of patterns (i.e., features) with respect to their distinctive characteristics.

Cadeco improves the feature selection method by adding a relevance variable for measuring how closely related a pattern is to a failing/passing execution [35]. This is done because standard feature selection does not distinguish between patterns that characterize one class (e.g., failing) from those that characterize the other class (e.g., passing). Two patterns can both have high feature scores but characterize different classes. We use the following function to calculate the relevance:

$$Relevance(e) = \begin{cases} +1, & \frac{n_f(e)}{n_f(e)+n_f(\overline{e})} > \frac{n_p(e)}{n_p(e)+n_p(\overline{e})} \\ -1, & \frac{n_f(e)}{n_f(e)+n_f(\overline{e})} \leq \frac{n_p(e)}{n_p(e)+n_p(\overline{e})} \end{cases}$$

Here, e denotes a bug pattern and $Relevance(e)$ determines to which class (i.e., fail or pass) e is closer. $\frac{n_f(e)}{n_f(e)+n_f(\overline{e})}$ is the ratio between the number of failing executions containing e and the total number of executions. A higher ratio indicates a stronger tie between e and the failing executions. $\frac{n_p(e)}{n_p(e)+n_p(\overline{e})}$ is the ratio between the number of passing executions containing e and the total number of executions. A higher ratio indicates a stronger tie between e and the passing executions. If $\frac{n_f(e)}{n_f(e)+n_f(\overline{e})}$ is greater than $\frac{n_p(e)}{n_p(e)+n_p(\overline{e})}$, e is closer to the failing executions, so the relevance score of e is $+1$. Otherwise, the relevance score of e is -1.

3.2.2 Cadeco Fault Localization Algorithm

Algorithm 1 shows Cadeco's feature selection algorithm for localizing faults. The algorithm takes the program P and a set of test inputs T as input and outputs a ranked list of concurrency bug patterns sorted by their likelihood to be faulty. P' is the instrumented program, SV is a set of shared variables, TR is a set of execution traces, and PTR is the predefined concurrency patterns. The algorithm executes each test input on P' to obtain traces (Line 2). Since the instrumentation point is inserted before and after each shared variable SV and the conditional synchronization operation S, the total number of execution traces is 2*$|T| * |SV + S|$. Next, the algorithm builds the coverage matrix to organize the data for feature selection (Line 4). It then runs a feature selection algorithm to compute the feature score for each pattern (Line 5). By using our enhanced algorithm, the feature score for a pattern

Algorithm 1 Concurrency fault localization using feature selection

```
Require: $P$, $T$
Ensure: $RL$
 1: $P$’ ← Instrument ($P$, $SV$)
 2: $TR$ ← execute $T$ on $P$’ to gather execution traces
 3: $PTR$ ← concurrency bug patterns
 4: $D$ ← BuildCovergeMatrix ($TR$, $PTR$)
 5: $S$ ← FeatureSelection ($PTR$, $D$)
 6: for each $p$ in $PTR$ do
 7:     $scfore \leftarrow S[p] \times Relevance(p)$
 8:     $M$.add($score$)
 9: end for
10: $RL$ ← sort($M$) return $RL$
```

is the product of the standard feature score and the relevance score of the pattern (Line 7). The relevance score is dynamically adjusted to reflect how close a pattern is to the failing execution.

The algorithm sorts patterns in descending order of feature scores (Line 10). Patterns that appear at the beginning of the ranked list are the ones with the highest feature scores among features that are closer to the failing class. Patterns that appear at the end of the ranked list are the ones with the highest feature scores among features that are closer to the passing class. Finally, the algorithm returns the ranked list of patterns.

3.3 Implementation

We implemented Cadeco Fault Localization in Java. The first component of Cadeco is the instrumentation and monitoring component, which is implemented on top of the Soot instrumentation framework [59]. Cadeco instruments the shared variables. In Java, shared variables can be identified by using thread escape analysis [33,50]; we adopted the conservative shared variable detection algorithm proposed previously [24] that uses the ThreadLocalObjectAnalysis API [20] provided by Soot [56] to compute variables that can potentially be read from and written to by multiple threads simultaneously. We also used the alias analysis [43] provided by Soot to identify a set of escaping variables that can potentially access the same location.

Cadeco injects artificial delays before and after each shared variable to increase the chance of exposing concurrency bugs. In the monitoring phase, Cadeco collects memory-access information generated from executing multiple threads. The accesses are then extracted for patterns [47]. To achieve this, we defined a fixed-size window for each shared-memory location. Every time there is an access (e.g., Read, Write) to the shared-memory location, it records and updates the window. When the window is full, and there is a new access to the shared-memory location, we extract the patterns from the window if one or more patterns are found. Then, we eliminate the oldest entry and shift the entries in the window, so that the new access can be recorded.

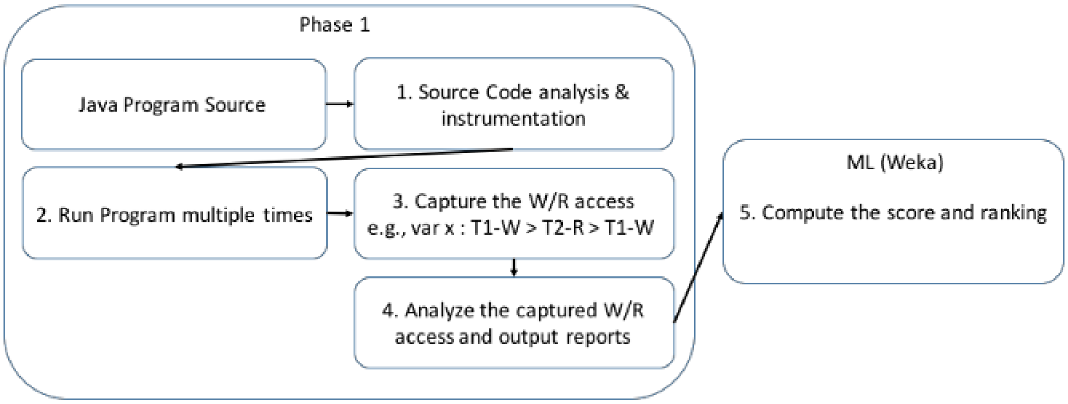

Phase 1
Java Program Source
1. Source Code analysis & instrumentation
2. Run Program multiple times
3. Capture the W/R access e.g., var x : T1-W > T2-R > T1-W
4. Analyze the captured W/R access and output reports
ML (Weka)
5. Compute the score and ranking

Figure 3.1: Implementation Overview

The second main component of Cadeco computes a feature score for each pattern (Section 3.2.1) and reports the list of ranked patterns in a text format. Feature selection is implemented using the Weka toolkit [19].

Cadeco is developed and implemented using several libraries and tools in Java. It is divided into several components and each of them serves different purposes and runs in different phases.

3.3.1 Source Code Sanitation and Re-formatting

To obtain more information from the source code, such as the line number and the line of code, we have to align the source code writing style, and remove unnecessary lines such as comments or dummy codes in the source code. We used an open source library "Java Comment Preprocessor (java-comment-preprocessor)" [26] to pre-process the original source code files and output them to a folder.

Then, we used the "Google Java Formatter (google-java-format)" [16] to format the source code, so that the coding style and the formatting of the source codes are consistent.

3.3.2 Locating Shared Variables

Since concurrency bugs usually involve shared variables, we would like to find out where the shared variables are located in the source codes. To achieve that, we extended a tool, "LEAP: Lightweight deterministic multiprocessor replay for concurrent Java programs" [36], to help locate the shared variables and retrieve details such as the line number, source file name, variable type, and the operation.

3.3.3 Source Code Instrumentation

In order to increase the diversity and the chance of bugs exposure, we perform static code instrumentation before running the analysis.

We have implemented a small utility with the help of an open source library namely "Java Parser (javaparser)" [25].

Since we have the sanitized version of the source code and the location of the shared variables, we can edit and instrument the source code file one by one. For each identified shared variable, we load up the sanitized source file. We then copy the content of the sanitized source file, add a sleep point before and after the shared variable line number, and output to a new file. The process generates two new files, which we use as the sources in the interleaving patterns capturing phase.

3.3.4 Interleaving Pattern Capturing

To find out the interleaving patterns, we implemented and extended the tool CalFuzzer, an extensible Active Testing Framework for Concurrent Programs. CalFuzzer depends on Soot, a framework for analyzing and transforming Java and Android Applications. We used the concept mentioned in the Falcon paper and implemented through CalFuzzer.

We added a new class to record several operations on the variables, such as read, write, wait, and notify. Each time there is an above-mentioned operation, the operation would be captured and stored in a defined window. Whenever that window is full, we would analyze the captured operations in the window, and check if they match any interleaving patterns (mentioned in the Falcon paper). We recorded down all the matched patterns, as they may be the possible patterns that caused the failures.

We run all the test cases of each test subject multiple times and stored the results into the database for report generation and input generation use.

3.3.5 Report Generation

In the next phase, we used our tool to retrieve the patterns from the database and analyze them. Based on different criteria and needs, we generated several sets of data and output them to a csv file. Examples of them are the interleaving patterns that appear in all cases and fail cases, interleaving def-use pattern, etc. Each report would list the patterns that are found suspicious, together with their scores and rankings. We used the formula from the Falcon paper as one of the scoring algorithms.

3.3.6 Input Generation for Feature Selection

In order to apply machine learning feature selection technique to locate the suspicious patterns, we need to convert our data to a format that the machine learning tool can recognize and support.

We used "WEKA" [62], one of the popular data mining tools in Java, to help us analyze and calculate the score of the patterns. We retrieved the data from the database and output them into a csv file that follows Weka input requirement.

3.3.7 Input Generation for Data Mining

Besides feature selection, we also applied data mining techniques to find suspicious interleaving patterns that may lead to fault. We applied frequent itemset mining technique with the help of "SPMF" [57], which is an open-source pattern mining library written in Java.

We retrieved the interleaving patterns of def-use pairs from the database, and output them into a csv file that follows SPMF input requirements. More will be discussed in Chapter 4.

3.3.8 Weka Data Mining

We used several feature selection algorithms. They are: Chi-Square Attribute evaluation (CH), Gain-Ratio Attribute evaluation (GR), Information-Gain Attribute evaluation (IG), Relief Attribute evaluation (RF), and Symmetrical Uncertainty Attribute evaluation (SU). Weka provided the above mentioned algorithms in their library. We applied the algorithms to suspicious interleaving patterns and obtained their rankings. Then, we consolidated the results and exported them into a csv file. The interleaving patterns with the highest ranking scores are listed in descending order.

3.3.9 Challenges

Bug Replication

We tried to use the test subjects mentioned in the related papers. However, some of the test subjects were either outdated, or unable to replicate the bug because the description was not detail enough or the information was incorrect. As a result, we had to debug the test subject ourselves, which was quite time consuming and difficult.

Bug Exposing

In order to increase the chance of exposing the bug, we run the benchmark programs with different parameters. For example, we increased the number of threads when running the program. We injected the sleep point before and after the shared variable, with different sleep time.

Tools dependencies

Our tool was integrated with multiple tools and libraries, which speeded up our development time. However, these libraries brought us problems when we tried to implement and glue them together. One of the problems was the version compatibility. For example, two of our tools, Calfuzzer and LEAP, depends on different versions of Soot, and Soot only supports Java 1.6 at the time of our development. On the other hand, the libraries we used to sanitize our source code, i.e. Java Comment Preprocessor and Google Java Formatter, required Java 8, and thus we could not put them to run altogether. To resolve this incompatibility issue, we had to separate the process into different phases, and run them one after one. This increased the overall run time.

Another issue was that there were some limitations in the frameworks in terms of functionality. We had to work around or enhance the function that was missing. Moreover, we found a few bugs in the frameworks that we had to fix ourselves or request the latest updates from the authors.

Verification

In order to know the occurrence of the bug, we need to understand the test subjects and their behaviors.

3.4 Evaluation

3.4.1 Objects of Analysis

We chose 10 open source benchmark programs, which are representative of real-world code and have been widely used in academic research. These included two objects downloaded from the Software-artifact Infrastructure Repository (SIR) [10], five objects from the Java Development Kit (JDK), and three larger objects [7,37,48]. The objects include both closed code units (code units equipped with test drivers) and open code units (libraries that require test drivers to close them).

Among the closed objects, Account is a banking program that simulates deposit and withdraw actions, Airline is an airline ticketing program, Pool is an Apache library that provides an object-pooling API and a number of object pool implementations, and Log4j is a Java logging application. Among the open objects, Cache4j is a cache for Java objects with a simple API and fast implementation. Arraylist, Bitset, HashMap, Hashtable, and Vector are synchronized collection classes provided by the JDK.

To address our questions, we required test inputs. Cadeco uses test inputs that are supplied with the programs. To increase the the diversity of coverage data involving concurrency violation patterns, Cadeco uses the instrumentation method described in Section 3.2.2 to collect execution traces.

Each of the benchmark programs contains a known concurrency bug. Each bug may be associated with one or more concurrency violation patterns.

We consider the following research questions:

RQ1: How effective is Cadeco at localizing concurrency faults?

RQ2: How efficient is Cadeco at localizing concurrency faults?

RQ3: How does Cadeco compare to a state-of-the-art approach?

RQ4: How does test case reduction impact the effectiveness of fault localization?

RQ5: How effective are different feature selection techniques at localizing concurrency faults?

RQ1 and RQ2 let us consider the overall effectiveness and efficiency of Cadeco in localizing concurrency faults. RQ3 lets us compare Cadeco to a baseline approach based on statistical analysis. RQ4 lets us compare the effectiveness of concurrency fault localization using different feature selection techniques. RQ5 lets us investigate whether reducing the number of tests runs impacts the effectiveness of concurrency fault localization.

3.4.2 Variables and Measures

This section describes the independent variable and the dependent variables.

Independent Variable

Our independent variable involves the fault localization techniques used in our study. We use Falcon as the baseline for comparison with our Cadeco. We chose Falcon because it is the most relevant to our approach. However, Falcon uses statistical fault localization (described in Section 2.1.1); it does not study the cost-effectiveness of using machine learning-based methods for handling high dimensional coverage data. Another spectrum-based fault localization technique for concurrent programs is CCI [29]. However, CCI gathers predicate entities rather than patterns, so it only ranks individual accesses. Previous study has also shown that Falcon outperformed CCI in terms of ranking faulty elements. Therefore, we did not compare Cadeco to CCI.

Within Cadeco, we consider five feature selection methods: Chi-Square (CH), Gain Ratio (GR), Information Gain (IG), Relief (RF), and Symmetrical Uncertainty (SU). By default, Cadeco uses Symmetrical Uncertainty (SU), which performs the best among all five methods. The implementations of these methods are available in Weka [19]. This controlled experiment allows us to evaluate the performance of different feature selection algorithms.

Within Cadeco, we also evaluate the effectiveness and efficiency of fault localization with reduced execution traces. Our goal is to determine whether the number of execution traces can be reduced to improve its efficiency with a minimal impact. While research has shown that the composition of the test suite impacts the effectiveness of fault-localization techniques [67], previous research focuses on sequential programs. To obtain a reduced set of traces, we filtered out and unified the traces that have the same set of patterns. For example, if Trace$_1$ and Trace$_2$ contain Pattern$_a$ and Pattern$_b$, we count them as one trace.

Dependent Variables

We choose metrics allowing us to answer each of our five research questions.

Ranking effectiveness To evaluate how accurately Cadeco localizes concurrency faults, we evaluate whether the correct answer (ground truth) is found and at which rank (for effectiveness) for each subject. We also evaluate the effectiveness by measuring the percentage of the program that must be examined to find the fault. This metric is called *expense* [67], which is computed by the following equation.

$$Expense = \frac{\overline{\sum_{i=1}^{n} Rank_{P_i}}}{\texttt{number of executable patterns}}$$

For a program that contains multiple faulty access patterns, we calculate the *Expense* by averaging the ranks of all faulty patterns. For example, if there is one faulty pattern ranked in 2^{nd} place among 100 patterns, the *Expense* is 0.02 (or 2%). If there are two faulty patterns ranked 2^{nd} and 3^{rd}, the the *Expense* is 0.025 (or 2.5%).

Ranking efficiency We measure the efficiency of Cadeco by computing the total time it took for fault localizations, including traces collection, features selection, and ranking calculation. When compared Cadeco to Falcon, the efficiency is almost identical because the two techniques spend the same time on collecting the traces. For Cadeco$_{redt}$, the efficiency is calculated using the total time of traces reduction, features selection, and ranking calculation.

3.4.3 Threats to Validity

The primary threat to external validity for this study involves the representativeness of our programs, bugs, and test cases. Other objects and test cases may exhibit different behaviors and cost-benefit tradeoffs. However, we do reduce this threat to some extent by using several varieties of well-studied open source code objects for our study, and test inputs and event interleavings generated by practical approaches. Since there are no standard benchmark suites for use in fault-localization of concurrency bugs, we used programs with known bugs.

The primary threats to internal validity for this study are possible faults in the implementation of our approach and in the tools that we used to perform evaluation. We controlled for this threat by extensively testing our tools and verifying their results against smaller programs for which we could manually determine the correct results. We also chose to use popular and established tools (e.g., Soot, Weka) for implementing the various stages of our approach.

Where construct validity is concerned, our measurements of effectiveness and efficiency of fault localization focus on the expense score and time required for analysis and test execution. However, other costs such as developer time can play a role in technique efficiency.

The faulty execution path that we found may not be a feasible path, as some of the details had been abstracted. However, it could still help developers and give better hints for them to locate the bugs in the program.

3.4.4 Results

We now analyze the results of our study relative to each of our research questions.

RQ1: Effectiveness

The goal of this study is to investigate how well our technique ranks patterns by determining whether highly ranked patterns correspond to true bugs. We ranked the benchmark programs with their default number of threads and inputs. To overcome nondeterminism, we executed each program 10 times.

Columns 2-7 in Table 3.1 summarize the results of the benchmark programs in terms of effectiveness. Column 2 reports the highest rank of the concurrency violation patterns using the feature selection technique (i.e., Symmetric Uncertainty) in Cadeco. Column 3 reports the number of patterns identified. Column 4 reports the number of patterns appearing in at least one failing execution. Column 5 reports the number of patterns appearing only in failing executions. Column 6 shows the highest rank of the first pattern found by Cadeco that corresponds to a true violation; the number in the parenthesis indicates the expense score. Column 7 reports the number of patterns involving the faulty memory accesses or synchronization operations. For example, the Account program has 31 patterns, among which 29 patterns appeared in at least one failing execution, 19 patterns appeared only in failing executions, the highest rank assigned by Cadeco to any pattern corresponding to a true bug was 1 with the expense score 25.8%, and the number of patterns related to the fault was 16.

As Table 3.1 shows, Cadeco ranked the true faulty pattern first in eight out of 10 programs. The results indicate that Cadeco *is effective at localizing concurrency faults*. On Arraylist, Cadeco ranked the faulty pattern at the second place. We initially suspected that the pattern ranked first was a concurrency fault that was accidentally discovered during traces collection. However, by further examining the data, we found that this pattern was related to a fault in the test driver, in which the method sequences used in some unit tests were incorrect. This led to failing executions even though the failures were not caused by a concurrency fault.

On program Pool, Cadeco ranked the true faulty pattern 8^{th}. Further investigation revealed that the top seven patterns are related to shared variable accesses (patterns 1-8 in Table 2.1), but the true pattern is a deadlock pattern. Nevertheless, these shared variable accesses ranked in the first place are actually synchronized by the wait-notify operations and thus related to the deadlock. Therefore, this pattern can still give insights for fault localization, if not directly.

RQ2: Efficiency

The goal of this research question is to investigate the efficiency of the Cadeco toolset by computing the total time spent on localizing the fault and slowdown caused by instrumentation. We computed the slowdown factor by comparing normal program execution and Cadeco-instrumented program execution.

Columns 8-9 in Table 3.1 summarize the results, showing the total execution time and the slowdown factor. Column 10 reports the number unique shared memory locations. For example, when executed with the Cadeco toolset, program Account spent 11 minutes on localizing the fault, including identifying shared variables, instrumentation, trace collection, running feature selection algorithms, and ranking. The slowdown over execution without any instrumentation was 2.0. The Account program has 13 shared memory locations.

Table 3.1: Results of RQ1 and RQ2: Effectiveness and Efficiency of Cadeco

Program	RQ1 (Effectiveness)						RQ2 (Efficiency)		
	High_s	# P	# P_f	# P_o	Rank	# F_p	Time	OH	# Loc
Account	0.516	31	29	19	1 (0.258)	16	11 mins	2x	13
Airline	0.576	12	9	2	1 (0.125)	1	16 mins	2.1x	9
ArrayList	0.117	122	74	35	2 (0.270)	15	39 mins	3x	232
BitSet	0.859	100	61	27	1 (0.140)	10	12 mins	2.2x	117
HashMap	0.94	133	114	93	1 (0.007)	9	26 mins	2.2x	92
Hashtable	0.981	51	30	7	1 (0.137)	5	393 mins	2.7x	158
Vector	0.522	65	17	13	1 (0.076)	5	48 mins	2.2x	70
Cache4j	0.364	50	50	15	1 (0.120)	4	38 mins	2.1x	42
Log4j	0.181	67	62	10	1 (0.287)	4	281 mins	2.8x	199
Pool	0.39	196	144	50	8 (0.275)	6	194 mins	2.7x	167

High_s=Highest suspiciousness score reported by the Symmetric Uncertainty algorithm. # P=Number of patterns identified. # P_f=Number of patterns appearing in at least one failure. # P_o=Number of patterns appearing in failing executions. Rank=Cadeco ranking (expense score). # F_p=Number of faulty patterns. OH=overhead of instrumentation. # Loc=Number of memory locations.

As the table shows, the time taken by Cadeco ranged from 11 minutes to 194 minutes, with an average of 105.8 minutes. The runtime overhead ranged from 2x to 3x, with an average of of 2.4x. These results indicate that Cadeco *is efficient at localizing concurrency faults in real world applications.*

RQ3: Comparison to Falcon

One of the most closely related studies of fault localization in concurrent programs with a ranking scheme is Falcon [47]. Thus, the goal of this research question is to compare the effectiveness and efficiency of Cadeco to Falcon.

For this study, we implemented Falcon in the Cadeco framework and applied it to our benchmark programs. Columns 2-7 of Table 3.2 report the results of the two techniques, including rank, time, and the number of traces. Since Falcon and Cadeco utilize the same set of traces, the efficiency score we used for comparison includes only the time spent on running the feature selection technique and ranking. For example, on program Account, both Cadeco and Falcon report the true faulty pattern at rank 1. However, Falcon takes more time than Cadeco.

As the table shows, Falcon tends to rank individual accesses of a real concurrency fault much lower than Cadeco in four out of 10 programs (i.e., ArrayList, Cache 4j, Log 4j, and Pool). Falcon was not able to localize faults in Pool because it does not handle deadlock. Also, Falcon takes 420% more time on average than Cadeco to analyze the traces and complete the ranking. This observation suggests that Cadeco

Table 3.2: RQ3 and RQ4: Comparing Cadeco to Falcon using Different Test Suites

Technique	Cadeco			Falcon			Cadeco$_{redt}$		
	Rank	Time	# traces	Rank	Time	# traces	Rank	Time	# traces
Account	1	531ms	180	1	1s	180	1	487ms	41
Airline	1	472ms	120	1	1s	120	2	408ms	8
ArrayList	2	2.3s	1150	5	5s	1150	2	688ms	176
BitSet	1	652ms	240	1	3s	240	1	508ms	73
HashMap	1	696ms	230	1	7s	230	1	539ms	52
Hashtable	1	27s	4190	1	2m34s	4190	1	801ms	91
Vector	1	1.4s	320	1	15s	320	1	1.3s	55
Cache4j	1	1.3s	600	9	10s	600	1	1.1s	391
Log4j	1	12s	4100	4	3m6s	4100	1	915ms	537
Pool	8	54s	3800	-	41s	3800	15	1.3s	91

Cadeco$_{redt}$=Reduced traces. Rank=Rank of the bug. Time=Process time. # traces=Number of traces analyzed.

is more effective and efficient at localizing concurrency faults than Falcon. The improvement is more obvious in larger programs.

RQ4: Impact of Test Case Reduction

The goal of this research question is to study whether reducing the number of traces can impact the effectiveness and efficiency of Cadeco. Columns 8-10 of Table 3.2 report the results of the reduction technique, including the rank, the time spent on reduction, data analysis, and ranking, and the ultimate number of traces.

As the table shows, Cadeco$_{redt}$ is less effective than Cadeco in terms of ranking in two out of 10 programs (i.e., Airline and Pool). However, Cadeco$_{redt}$ improved the efficiency over Cadeco on all benchmark programs, ranging from 7.69% to 4053.84%, with 887.84% on average. The number of traces was reduced by a range of 34.83% to 97.82%, with an average of 80.22%. These results indicate that *test case reduction can slightly affect the effectiveness of concurrency fault localization but can improve its efficiency.*

RQ5: Different Feature Selection Techniques

Table 3.3 shows the effectiveness of different feature selection methods in terms of ranking scores. As the table shows, on four programs (i.e., Account, ArralyList, HashMap, and HashTable), the five techniques are equally effective. On the other programs, the Symmetric Uncertainty (SU) technique and Information Gain (IG) were better than at least one of the other four techniques on all benchmark programs. Although SU and IG have the same average ranking score, based on the overall performance and the results we analyzed, SU is slightly better than IG. In contrast, Relief (RF) was the worst choice for concurrency fault localization in our experiment.

Table 3.3: RQ5: Using Different Feature Selection Techniques

Program	Chi-Square	Gain Ratio	Info. Gain	Relief	Sym. U.
Account	1	1	1	1	1
Airline	2	1	2	3	1
ArrayList	2	2	2	2	2
BitSet	1	1	1	3	1
HashMap	1	1	1	1	1
Hashtable	1	1	1	1	1
Vector	1	1	1	5	1
Cache4j	3	3	1	1	2
Log4j	1	4	1	1	1
Pool	8	8	8	12	8
Average	2.1	2.3	1.9	3.0	1.9

Info. Gain=Information Gain Sym. U.=Symmetric Uncertainty

If we generalize these results with other subjects, we can conclude that *when using feature selection to localize concurrency faults, Symmetric Uncertainty is the best choice, whereas Relief is the worst.*

3.5 Discussion and Summary

3.5.1 Discussion

We demonstrated in the previous section that Cadeco is useful for localizing concurrency faults. Specifically, we showed (subject to stated threats to validity) that 1) Cadeco is more effective and efficient at localizing concurrency faults than a state-of-the-art spectrum-based concurrency fault localization technique; 2) Reducing the number of traces can impact the effectiveness of fault localization to a small extent, but can improve its efficiency; and 3) Symmetric Uncertainty (SU) is the most effective feature selection technique in localizing concurrency faults.

Our results have implications for practitioners and researchers, as discussed below.

Implications for Practitioners

Results indicate that certain feature selection techniques are more effective and efficient than statistical analysis techniques in concurrency fault localization. Practitioners can apply this finding by collecting coverage data involving concurrency violation patterns and using the Symmetric Uncertainty (SU) technique to localize the root cause (i.e., patterns) of the concurrency fault. In addition, our results showed that test case reduction can improve the efficiency of Cadeco while affecting the effectiveness only to a small degree. Therefore, in relation to large amounts of coverage data, practitioners can reduce the number of execution traces to achieve higher efficiency.

Implications for Researchers

The use of coverage criterion. Our study uses existing test inputs against the program under different interleavings to obtain execution traces. While the initial study on the def-use coverage criterion is promising, further research is needed to study the effectiveness and efficiency of fault localization under different concurrency coverage criteria [38].

Localizing multiple faults. Our study focuses on concurrency faults only. It might be useful to search for both concurrency faults and sequential faults simultaneously. In addition to concurrency access patterns, more information (e.g., statements) might be needed for the coverage data. Both concurrency access patterns and sequential elements can be formulated into features. Further research is needed to investigate what additional information should be provided to developers for locating both sequential and concurrency faults in a cost-effective way.

3.5.2 Summary

In this work, we propose an automated approach for fault localization in concurrent programs using feature selection. Cadeco fault localization consists of two steps. In the first step, Cadeco employs dynamic analysis to identify patterns associated with concurrency violation patterns. In the second step, Cadeco ranks these patterns using feature selection methods based on passing and failing executions. Our empirical study shows that Cadeco fault localization is highly effective in ranking true fault patterns and efficient in storage and time overheads. Among five feature selection methods, Symmetric Uncertainty (SU) is the most effective. Comparing to the state-of-the-art spectrum-based concurrency fault localization techniques, Cadeco fault localization is more effective in 40% of the benchmark programs and performs equally well in the rest. We also study the impact of using test case reduction technique; the results indicate that the impact of reducing test cases based on shared variable coverage is negligible.

Chapter 4 Context-Aware Debugging for Concurrent Programs

4.1 Introduction

In Chapter 3, we presented our technique on localizing the root cause of a concurrency fault by ranking suspicious concurrency access patterns. Since our goal is to help developers better understand the concurrency bug by providing possible faulty execution paths within and across different threads, we present our approach and details in this chapter. To analyze and reconstruct a faulty execution path, there are two basic steps. First, we use a frequent itemset data mining algorithm to extract the most frequent interleaving patterns in the failed executions. Then, we reconstruct the faulty execution path from the program's inter-thread inter-procedural control flow graph by searching for paths that contain the mined frequent patterns.

We conduct an empirical study on ten real-world concurrent programs. The results showed that our technique can find precise execution paths with low cost.

4.2 Approach

In this section, we describe the two steps of our approach in detail.

4.2.1 Frequent Concurrency Access Pattern Mining

We obtained the failing execution traces from the traces collected in Chapter 3. In addition to the inter-thread def-use pairs, we collected the intra-thread def-use pairs. Using these two kinds of def-use paris, we could guide the reconstruction of paths within individual threads. Specifically, we monitored and recorded the read and write accesses of the shared variables. We defined a fixed-size window to store the read write accesses and extracted the pattern once the window is full. If the extracted patterns fulfill the inter-thread def-use pattern, e.g., Write in Thread 1 Read in Thread 2 ($\mathrm{W_1R_2}$) or intra-thread def-use pattern, e.g., Write in Thread 1 Read in Thread 1 ($\mathrm{W_1R_1}$) (subscript is the thread id), we saved them for later stage.

Table 4.1 indicates the inter-thread and intra-thread def-use pairs collected from two test executions. Row 2 column 1 shows an example of an inter-thread def-use pair. The record $W1\#Account.java\#9 > R2\#Account.java\#14$ can be interpreted as: There is a Write operation in Thread 1 in Account.java line 9 (which is a def) and a Read operation in Thread 2 in Account.java line 14 (which is a use) on a shared variable. Row 5 column 1 shows an example of an intra-thread def-use pair. The record $W1\#Account.java\#9 > R1\#Account.java\#14$ can be interpreted as: There is a Write operation in Thread 1 in Account.java line 9 (which is a def) and a Read operation in Thread 1 in Account.java line 14 (which is a use) on a shared variable.

Table 4.1: Example of interleaving patterns for import

testcase1	testcase2
W1#Account.java#9 > R2#Account.java#14	W1#Account.java#9 > R2#Account.java#14
W1#Main.java#18 > R2#Main.java#52	W1#Main.java#18 > R2#Main.java#52
W1#Account.java#9 > R2#Account.java#14	W1#Account.java#18 > R2#Account.java#22
W1#Account.java#9 > R1#Account.java#14	W1#Account.java#9 > R1#Account.java#14

Next, we use the frequent itemset mining algorithm [17] to extract frequent interleaving def-use patterns within threads and across threads. The patterns are used to guide the search for faulty paths. Since the frequent itemset mining algorithm only takes in matrices, we need to convert the def-use pairs into a matrix by assigning a unique id to each of them. Specifically, we build a matrix where each row represents a failure run and each column is mapped to an unique id. Table 4.2 shows an example of the inputs to FP-Max converted from the patterns in Table 4.1. When applying the frequent itemset mining algorithm, we use a support value 40%, which is a general threshold value. This indicates that the itemsets have occurred in 40% of the total traces are retrieved.

Table 4.2: Example of interleaving patterns for the process of FP-Max

testcase 1	2	4	6	8	9	15
testcase 2	2	4	7	9	12	15

Algorithm 2 shows the Cadeco's frequent itemset mining algorithm. The algorithm takes the interleaving def-use pairs (DUP) and support (S) as the inputs and outputs a list of frequent patterns (FP). It first converts the def-use pairs into a matrix (M) (Line 1). Then, it finds the frequent itemset within M based on S (Line 2). Next, it returns a list of frequent itemset (R) (frequent def-use pairs) in mapping id format. We convert the mapping ids back to a list of frequent patterns FP (Line 3). Table 4.3 shows the parsed result after converting from pattern id to the patterns in text.

Algorithm 2 Frequent Itemset Mining

```
Require: $DUP$, $S$
Ensure: $FP$
$M \leftarrow$ ConvertToMatrix ($DUP$)
$R \leftarrow$ FP-Max ($M$, $S$)
$FP \leftarrow$ ConvertToPattern ($R$) return $FP$
```

Table 4.3: Example of interleaving patterns FP-Max parsed results

freq. patterns 1
W1#Account.java#9 > R2#Account.java#14
W1#Main.java#18 > R2#Main.java#52
W1#Account.java#9 > R1#Account.java#14

4.2.2 Path Reconstruction

To reconstruct execution paths, we first build an inter-thread inter-procedural control flow graph (IICFG). Each node in the graph represents a piece of a code block, which consists of a sequence of code statements and the corresponding thread id. We need a few pieces of information to build the IICFG, such as the CFG and the call graph of the program and we use Soot to extract them. The execution of Soot is separated into several phases. To achieve that, we implement a transformer and run it in a particular phase in Soot. In Soot, a transformer is an object that transforms some block of code to some other block of code. By implementing our own transformer, we can retrieve the CFG and the call graph information of the program. A CFG is made up of a set of different block graphs. We use the API provided by Soot to obtain all the block graphs, starting from the entry point of the program to the exit point of the program. Then, using the call graph, we eventually follow the flow of how the methods are being called and extract the code blocks of the methods one by one. When we run Soot on the test programs, we specify our transformer in the phase so that Soot will automatically go through that phase during runtime. As a result, we retrieve the CFG of the methods together with other corresponding information. Since Soot cannot provide any information other than a method based CFG, we output the CFG information of all visited methods to a file and use it to build the IICFG later.

Algorithm 3 Path Reconstruction

```
Require: $CFG$, $CG$, $FIP$
Ensure: $Path$
$IICFG \leftarrow$ buildIICFG ($CFG$, $CG$, $FIP$)
for each $fip$ in $FIP$ do
    $n \leftarrow$ BFS ($IICFG$, $fip$)
    $Path \leftarrow$ matchAndColor ($n$, $fip$)
end for
return $Path$
```

Algorithm 3 shows the abstract of our path reconstruction algorithm. We build the ICFG using the graph data structure. Starting from the root node, which is the entry point of the program, we find the neighbor nodes and insert them into the graph. After all the nodes are inserted into the graph, we connect the nodes by adding edges between them. The constructed graph is a directed graph that illustrates the flow of the program. We build two of the same directed graphs and assign a thread id to each of them to simulate the multi-thread environment. Lastly, we attach the two ICFGs as the children of a root node and the IICFG is created.

Once the IICFG is created, we use the frequent concurrency access patterns - the results we obtained from the frequent itemset mining to guide searching for faulty paths from the IICFG. We traverse the graph using breadth-first search. Each node of the graph is the code block containing the line number and the name of the source code. An item in the frequent itemset pattern contains a def-use pair. When a def or use in the frequent itemset pattern is found in the current node, we highlight it with a color. We traverse the graph from the root to the end. When it is done, all the nodes in the graph that contains the frequent itemset pattern would be highlighted with a color. We can then find the path through the connected nodes and edges. We output the IICFG to an image for visual verification.

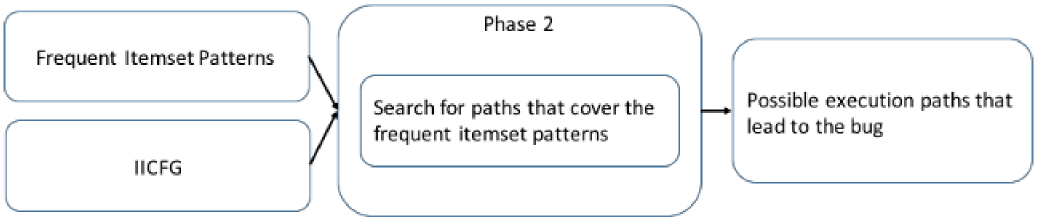

Frequent Itemset Patterns
IICFG
Phase 2
Search for paths that cover the frequent itemset patterns
Possible execution paths that lead to the bug

Figure 4.1: Context-Aware overview

4.3 Implementation

We implemented our approach in Java.

4.3.1 Frequent Itemset Mining

The frequent itemset mining (FP-Max) algorithm is implemented and provided as a library in SPMF [57]. In order to use it, we need to convert our data into the desired format and save in a csv file. We map the interleaving patterns with a unique id. During the conversion, the mapping between the pattern and id is stored in a mapping file. We run the SPMF application in the command line with the corresponding arguments. Once SPMF has finished running, it outputs the results in a csv file. We convert the content of the csv file back into the interleaving patterns using the mapping file. Table 4.3 is an example of the final output.

4.3.2 Constructing IICFG

To construct the IICFG, we implement a phase in Soot by extending the `SceneTransformer`. Since Soot has limited functions and cannot export the ICFG of the programs, we need to implement our own class to extract the CFG and build the ICFG and IICFG with the IICFG builder, which is another tool that we wrote. We add functions in the SceneTransformer, so that it traverses the program and extracts the block information of the methods at runtime. Then, we export the code block graph and the corresponding information such as line number, branches, external method calls of the test subjects. Once we have the code block graph information exported, we import them into the IICFG builder. The IICFG builder, written in Java 8, uses an open source library "JGraphT" [27] to generate the ICFG and IICFG, based on the code block graph exported by our SceneTransformer.

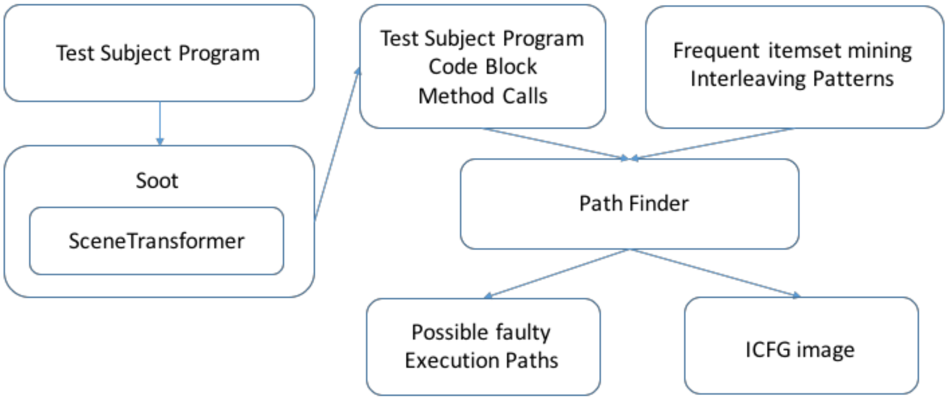


Figure 4.2: Implementation Overview

While creating the IICFG, we fill in the frequent itemset interleaving patterns to add the cross edges.

Then, we export the graph in "DOT" format, which is a graph description language, and use graphviz [15] to convert the "DOT" file into an image.

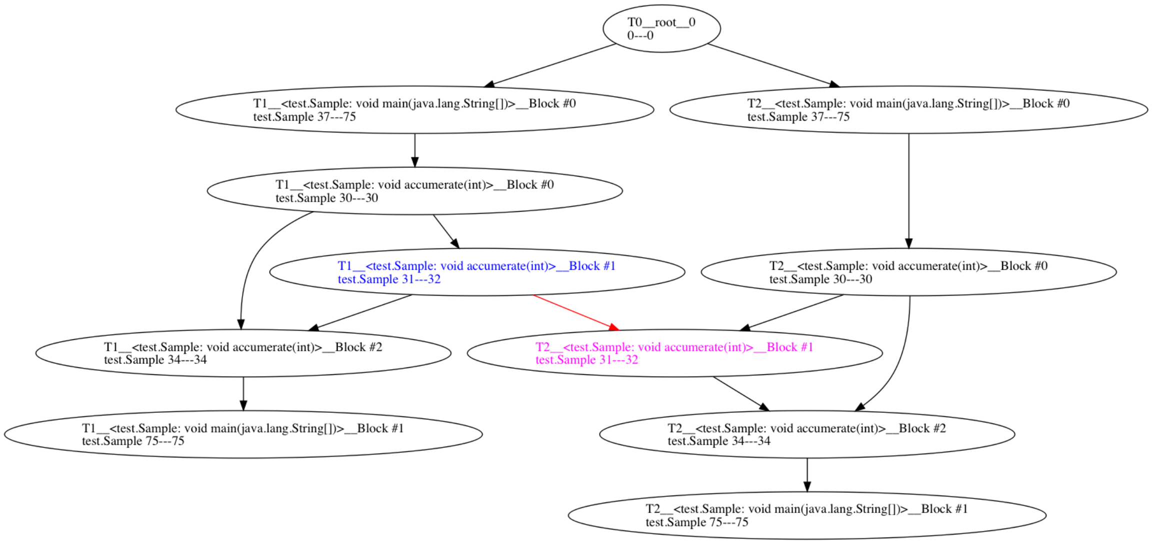

T0__root__0
0---0
T1__<test.Sample: void main(java.lang.String[])>__Block #0
test.Sample 37---75
T2__<test.Sample: void main(java.lang.String[])>__Block #0
test.Sample 37---75
T1__<test.Sample: void accumerate(int)>__Block #0
test.Sample 30---30
T1__<test.Sample: void accumerate(int)>__Block #1
test.Sample 31---32
T2__<test.Sample: void accumerate(int)>__Block #0
test.Sample 30---30
T1__<test.Sample: void accumerate(int)>__Block #2
test.Sample 34---34
T2__<test.Sample: void accumerate(int)>__Block #1
test.Sample 31---32
T1__<test.Sample: void main(java.lang.String[])>__Block #1
test.Sample 75---75
T2__<test.Sample: void accumerate(int)>__Block #2
test.Sample 34---34
T2__<test.Sample: void main(java.lang.String[])>__Block #1
test.Sample 75---75

Figure 4.3: Example output of the IICFG with the interleaving patterns
Block in blue is a write operation, while block in cyan is a read operation.
Red edge indicates the possible data flow that leads to the bug.

Figure 4.3 shows an IICFG of a sample program. The sample program initializes two threads, i.e. T1 and T2. There is a shared variable defined (i.e. write operation) at T1 test.Sample class, accumerate(int) method, block #1 (block in blue color) and being used (i.e. read operation) at T2 test.Sample class, accumerate(int) method, block #1 (block in cyan color). Since this is an inter def-use, it would be a potential cause of an interleaving issue. The red edge indicates the possible data flow (def-use operation) that may lead to the bug.

4.3.3 Challenges

First, there were issues when generating the IICFG of the test subjects due to the limitation of the functionality and the bugs in the Soot framework, which required extra time to investigate the cause of the issues and resolved them. Secondly, we needed extra care when handling special cases such as loop and recursive calls, such that the graph would not fall into an endless loop. Thirdly, some of the generated IICFG were so large that contains 1000+ nodes, which would look complicated and difficult to follow. Therefore, we had to minimize the number of nodes by abstracting some of them and removing some unnecessary information, such as unused nodes or unrelated methods. Fourthly, we had to match the line number of the source code and the interleaving patterns. Whenever there are changes in the source code, we have to re-run the whole process, which takes quite a lot of time.

4.4 Evaluation

We evaluate our approach and results in this section.

4.4.1 Objects of Analysis

We used the results of the 10 open source benchmark programs from Cadeco Fault Localization phase. Since each of the benchmark programs contains a known concurrency bug, we could gather the interleaving faulty patterns from the failing executions.

We consider the following research questions:

RQ1: How effective is Cadeco at extracting the possible execution paths?

RQ2: How efficient is Cadeco at extracting the possible execution paths?

4.4.2 Variables and Measures

Independent Variable

There are several independent variables in our experiment. The support value used in frequent itemset mining is one of them. The CFG extracted from the program remains the same every time we execute the tests.

Dependent Variables

The result of the frequent itemset mining depends on several factors: the support value we feed into the frequent itemset mining algorithm; the interleaving patterns captured; and filtered during Cadeco fault localization phase. Since support is an indication of how frequently the itemset occurs in the dataset, changing the support may give a different result set.

4.4.3 Threats to Validity

The primary threat to external validity is the abstraction when constructing the IICFG and searching for the possible faulty execution path. In a complex program, the IICFG is huge, and it is difficult to search paths within the graph. Thus, we try to give an abstraction of the ICFG with a few strategies such as ignoring or skipping unrelated nodes and visiting the node details only if a pattern exists within them. We also noticed that some of the paths were not feasible as false positive paths may be found in the graph. A path may be valid in the graph perspective, but not in a program flow. We also assumed there is only one shared variable per line. However, multiple shared variables could be found on the same line. Thus, the interleaving patterns may not be pointing to the same shared variable.

4.4.4 Results

Table 4.4: Frequent Intra and Inter Def-Use Pairs

Test subj.	Intra-DU	Inter-DU	Fq. Intra-DU	Fq. Inter-DU	% Code Exam.	# Faulty Paths
Account	6	7	6	3	65%	2
Airline	14	2	13	1	79%	2
ArrayList	13	15	10	4	3%	6
BitSet	19	13	14	4	14%	6
HashMap	40	15	13	11	2%	4
Hashtable	21	21	20	10	3%	5
Vector	32	11	14	1	34%	3
Cache4j	7	14	7	6	4%	5
Log4j	30	53	30	23	5%	8
Pool	129	62	48	5	24%	6

Test subj.=Test Subjects
Intra-DU=Number of Intra def-use pairs
Inter-DU=Number of Inter def-use pairs
Fq. Intra-DU=Number of Freq. Intra def-use pairs
Fq. Inter DU=Number of Freq. Inter def-use pairs
% Code Exam.=Percentage of code examined
# Faulty Paths=Number of possible faulty execution paths

Table 4.4 shows the frequent intra and inter def-use pairs that our approach has identified for each test subject program before and after frequent itemset mining. There is an average of 75.2% reduced intra def-use pairs and 37.4% reduced inter def-use pairs. This helps narrows down the number of patterns that we need to analyze and examine. Column 6 shows the percentage of code examined in order to retrieve the possible faulty execution paths. The lower the percentage, the more efficient it is. Column 7 shows the summary of the results, including the number of possible faulty paths we have identified. Take the test subject *Account* as an example: we have identified two possible faulty paths that may lead to the bug. By following the path suggested by Cadeco and reading the source code of the program, it turned out that the problem exists and the path is correctly guiding the right direction to the problem. For some of the larger programs, there are more of the possible faulty paths. We have to examine them one by one as some of the paths may not be a valid or feasible execution path in the program.

We identify the possible faulty execution paths from the generated output of the IICFG. We search for the paths starting from the root node, which is the starting point of the program. We color the nodes (i.e. code block) that contain the line of the interleaving patterns while transversing the graph. Afterwards, we connect all the nodes that match and find paths that could pass through them all.

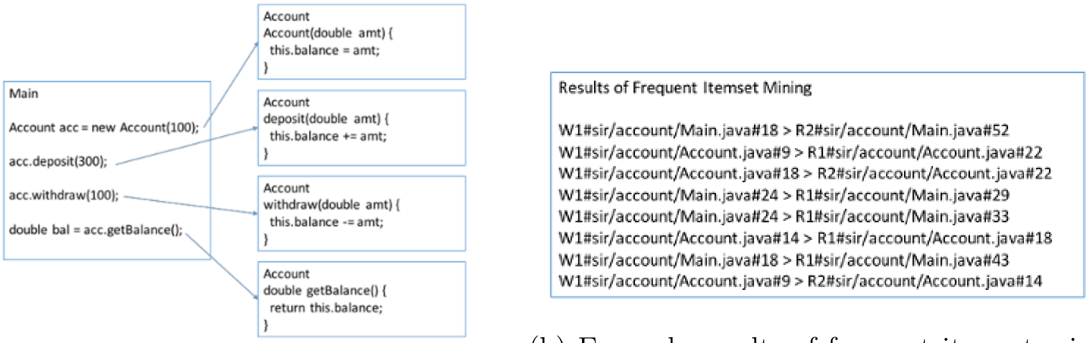
Table 4.5: Time needed for generating the results

Test subjects	Freq. Itemset Mining	CFG Extraction	IICFG Construction
Account	11ms	63s	5s
Airline	13ms	66s	6s
ArrayList	24ms	70s	7s
BitSet	16ms	68s	7s
HashMap	70ms	69s	6s
Hashtable	71ms	69s	8s
Vector	22ms	70s	7s
Cache4j	121ms	91s	32s
Log4j	188ms	102s	51s
Pool	212ms	119s	49s

Freq. Itemset Mining=Time used for Freq. Itemset Mining in seconds
CFG Extraction=Time used for CFG Extraction
IICFG Construction=Time used for IICFG Construction

Table 4.5 shows the time needed when running the frequent itemset mining, CFG extraction, and IICFG construction. The time for frequent itemset mining is proportional to the number of patterns identified. Also, for programs that have larger code base, the time needed to extract the CFG and construct the ICFG would increase accordingly.

An example of how to construct the control flow graph of a test subject is illustrated below:



```
Main

Account acc = new Account(100);

acc.deposit(300);

acc.withdraw(100);

double bal = acc.getBalance();
```

```
Account
Account(double amt) {
  this.balance = amt;
}
```

```
Account
deposit(double amt) {
  this.balance += amt;
}
```

```
Account
withdraw(double amt) {
  this.balance -= amt;
}
```

```
Account
double getBalance() {
  return this.balance;
}
```

(a) Example code snippet (Account)

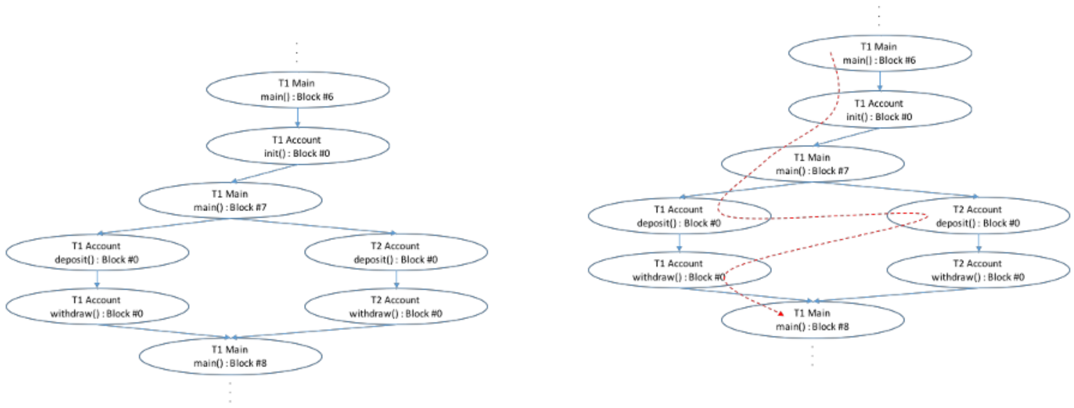
```
Results of Frequent Itemset Mining

W1#sir/account/Main.java#18 > R2#sir/account/Main.java#52
W1#sir/account/Account.java#9 > R1#sir/account/Account.java#22
W1#sir/account/Account.java#18 > R2#sir/account/Account.java#22
W1#sir/account/Main.java#24 > R1#sir/account/Main.java#29
W1#sir/account/Main.java#24 > R1#sir/account/Main.java#33
W1#sir/account/Account.java#14 > R1#sir/account/Account.java#18
W1#sir/account/Main.java#18 > R1#sir/account/Main.java#43
W1#sir/account/Account.java#9 > R2#sir/account/Account.java#14
```

(b) Example results of frequent itemset mining (Account)

Figure 4.4: Requirements for building IICFG

Figure 4.4a shows the code snippet of one of the test subjects *Account*. In the *main* method, there are four external method calls from the *Main* class to *Account* class. They are: the *constructor*, *deposit*, *withdraw* and the *getBalance*.

Figure 4.4b shows the results after applying frequent itemset mining. They are inter and intra def-use pairs captured in the failure executions.



(a) Snippet IICFG of Account

(b) Snippet IICFG of Account with faulty data flow path in dotted line

Figure 4.5: Example of IICFG

Figure 4.5a and 4.5b shows the snippet ICFG and the snippet of ICFG with data flow path.

The program spawns two threads that would execute *deposit* and *withdraw* concurrently. Since the shared variable in the *Account* class is not protected by the *synchronized* keyword, the possible faulty data flow path is found.

4.5 Discussion and Summary

4.5.1 Discussion

As demonstrated in the previous section, our technique searches possible faulty execution paths to help developers debug. We show the context of the possible faults such as the source code, method calls, and the line numbers that may lead to the bug. However, further study and tests should be conducted in order to improve the efficiency and accuracy of the approach.

4.5.2 Summary

In this work, we propose an approach to look for possible bugs by using frequent itemset mining from faulty interleaving pattern traces and the inter-procedural control flow graph of the programs. Based on the interleaving patterns captured by Cadeco in the fault localization phase and the code blocks that contain them in the IICFG, we can search for the abstract faulty execution flow paths. Developers could use the paths to locate or narrow down the root cause of the bug from the concurrent programs.

Chapter 5 Conclusion and Future Work

5.1 Conclusion

We present Cadeco, a framework to assist developers in localizing concurrency bugs by providing them the highly rated interleaving patterns, and the possible faulty execution paths. The aim of our tool is to help developers understand and find out the cause of the concurrency issues. There are two main important works in Cadeco: 1. Capture the interleaving patterns and identify the possible faulty interleaving patterns using a machine learning feature selection technique. 2. Apply frequent itemset pattern mining to find the most frequent faulty interleaving patterns and simulate the possible execution paths that lead to the bug through IICFG. By combining these two works, we are able to retrieve and locate the highly suspicious concurrent bugs.

5.2 Improvement in Fault Localization

As future work, we will investigate the cost-effectiveness of fault localization under different interleaving criteria. We will also study our techniques on localizing multiple faults including both sequential and concurrency faults. In order to improve and increase the accuracy of Cadeco results, we can increase the number of analysis samples and gather more traces by running more test subjects. We can also increase the diversity and occurrence of the bug by running each test case multiple times in each test subject and adjusting the schedule sleep time when instrumenting the test subject, so that the bugs can be exposed in a higher chance. In addition, since the number of running threads can affect the interleaving patterns captured in the window, we can vary the window size when collecting and analyzing the operations (i.e., read, write) on the shared variables, such that we may extract more different patterns. We plan to perform additional experiments on more diverse subjects to determine whether our technique generalizes.

5.3 Improvement in Context-Aware Debugging

There is much work to do in order to improve the result and practical usage.
The current algorithm only gives an overview and an abstract manner of the possible faulty execution paths. It would be better to improve the algorithm to give a more concise faulty execution paths.

There are a few features that would improve the usability of Cadeco. First, current version of Cadeco works under command console. It would be more convenience if it becomes a plugin for an IDE tool. Second, a guided search on the possible faulty execution paths would be helpful as developers could look into the details step by step. Third, an auto search function that finds and suggests the faulty execution paths would be a helpful feature.

Finally, we will perform a user study to evaluate the effectiveness of Cadeco, in order to help us further improve the performance of Cadeco in helping developers localize concurrency bugs.

Bibliography

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[2] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 89–98. IEEE, 2007.

[3] R. Agarwal and S. D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. pages 51–60, 2006.

[4] C. Artho, K. Havelund, and A. Biere. High-level data races. *Softw. Test. Verif. Rel.*, 13:207–227, 2003.

[5] A. L. Blum and P. Langley. Selection of relevant features and examples in machine learning. *Artificial intelligence*, 97(1):245–271, 1997.

[6] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: proportional detection of data races. In *PLDI*, pages 255–268, 2010.

[7] http://cache4j.sourceforge.net.

[8] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan. Identifying bug signatures using discriminative graph mining. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 141–152. ACM, 2009.

[9] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *IEEE 31st International Conference on Software Engineering*, pages 34–44, 2009.

[10] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. 10(4):405–435, 2005.

[11] R. O. Duda, P. E. Hart, D. G. Stork, et al. *Pattern classification*, volume 2. Wiley New York, 1973.

[12] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm. Ifrit: Interference-free regions for dynamic data-race detection. In *OOPSLA*, pages 467–484, 2012.

[13] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, pages 256–267, 2004.

[14] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *PLDI*, pages 121–133, 2009.

[15] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[16] Google Java Format. https://github.com/google/google-java-format.

[17] G. Grahne and J. Zhu. High performance mining of maximal frequent itemsets. In *6th International Workshop on High Performance Data Mining*, 2003.

[18] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

[19] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1):10–18, 2009.

[20] R. L. Halpert, C. J. F. Pickett, and C. Verbrugge. Component-based lock allocation. pages 353–364, 2007.

[21] K. Havelund. Using runtime analysis to guide model checking of Java programs. In *Proceedings of the International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 245–264, 2000.

[22] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[23] H.-Y. Hsu, J. A. Jones, and A. Orso. Rapid: Identifying bug signatures to support debugging activities. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 439–442. IEEE Computer Society, 2008.

[24] J. Huang, P. Liu, and C. Zhang. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs. In *FSE*, pages 207–216, 2010.

[25] Java Parser and Abstract Syntax Tree. https://github.com/javaparser/javaparser.

[26] Java Comment Preprocessor. https://github.com/raydac/java-comment-preprocessor.

[27] JGraphT. https://github.com/jgrapht/jgrapht.

[28] L. Jiang and Z. Su. Context-aware statistical debugging: From bug predictors to faulty control flow paths. 2007.

[29] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *ACM Sigplan Notices*, volume 45, pages 241–255, 2010.

[30] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, 2005.

[31] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. pages 110–120, 2009.

[32] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang. Static data race detection for concurrent programs with asynchronous calls. In *FSE*, pages 13–22, 2009.

[33] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. pages 226–239, 2007.

[34] B.-C. Kim, S.-W. Jun, D. J. Hwang, and Y.-K. Jun. Visualizing potential deadlocks in multithreaded programs. pages 321–330, 2009.

[35] T.-D. B. Le, D. Lo, and M. Li. Constrained feature selection for localizing faults. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 501–505, 2015.

[36] LEAP: Lightweight deterministic multiprocessor replay for concurrent Java programs. https://www.cse.ust.hk/prism/leap/.

[37] http://logging.apache.org/log4.

[38] S. Lu, W. Jiang, and Y. Zhou. A study of interleaving coverage criteria. In *The 6th joint meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers*, pages 533–536, 2007.

[39] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339, 2008.

[40] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting atomicity violations via access interleaving invariants. pages 37–48, 2006.

[41] B. Lucia, B. P. Wood, and L. Ceze. Isolating and understanding concurrency errors using reconstructed execution fragments. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 378–388, 2011.

[42] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. In *PLDI*, pages 134–143, 2009.

[43] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. pages 308–319, 2006.

[44] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. pages 167–178, 2003.

[45] S. Park, M. J. Harrold, and R. Vuduc. Griffin: grouping suspicious memory-access patterns to improve understanding of concurrency bugs. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 134–144. ACM, 2013.

[46] S. Park, R. Vuduc, and M. J. Harrold. A unified approach for localizing non-deadlock concurrency bugs. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 51–60, 2012.

[47] S. Park, R. W. Vuduc, and M. J. Harrold. Falcon: Fault localization in concurrent programs. In *ICSE*, pages 245–254, 2010.

[48] https://commons.apache.org/proper/commons-pool.

[49] S. Roychowdhury and S. Khurshid. Software fault localization using feature selection. In *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering*, pages 11–18, 2011.

[50] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. pages 12–23, 2001.

[51] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 56–66, 2009.

[52] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav. Testing atomicity of composed concurrent operations. In *OOPSLA*, pages 51–64, 2011.

[53] N. Shahmehri, M. Kamkar, and P. Fritzson. Semi-automatic bug localization in software maintenance. In *Software Maintenance, 1990, Proceedings., Conference on*, pages 30–36. IEEE, 1990.

[54] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng. Do i use the wrong definition?: Defuse: Definition-use invariants for detecting concurrency and sequential bugs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 160–174, 2010.

[55] A. Silberschatz, P. Galvin, and G. Gagne. *Applied Operating System Concepts.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 2001.

[56] Soot, a framework for analyzing and transforming java and android applications, 2016. https://sable.github.io/soot/.

[57] SPMF is an open-source data mining library. https://www.philippe-fournier-viger.com/spmf/.

[58] R. Tzoref, S. Ur, and E. Yom-Tov. Instrumenting where it hurts: an automatic concurrent debugging technique. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 27–38, 2007.

[59] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, page 13, 1999.

[60] L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *PPOPP*, pages 137–146, 2006.

[61] W. Wang, Z. Wang, C. Wu, P.-C. Yew, X. Shen, X. Yuan, J. Li, X. Feng, and Y. Guan. Localization of concurrency bugs using shared memory access pairs. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 611–622, 2014.

[62] Weka 3: Data Mining Software in Java. https://www.cs.waikato.ac.nz/ml/weka/.

[63] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[64] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[65] X. Xie, W. E. Wong, T. Y. Chen, and B. Xu. Spectrum-based fault localization: Testing oracles are no longer mandatory. In *Quality Software (QSIC), 2011 11th International Conference on*, pages 1–10, 2011.

[66] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, pages 1–14, 2005.

[67] Y. Yu, J. A. Jones, and M. J. Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *Proceedings of the 30th international conference on Software engineering*, pages 201–210, 2008.

[68] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[69] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. Conseq: detecting concurrency bugs through sequential errors. In *ACM SIGPLAN Notices*, volume 46, pages 251–264, 2011.

[70] W. Zhang, C. Sun, and S. Lu. Conmem: Detecting severe concurrency bugs through an effect-oriented approach. pages 179–192, 2010.

[71] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 33–42. ACM, 2005.

Vita

Justin Chu

Education

BSc Computer Science, University of Manchester, United Kingdom, July 2004

Professional Experience

Graduate Research Assistant, Computer Science Department, University of Kentucky, Lexington, Kentucky. May 2016 - Aug 2017.

Senior Software Engineer, ASTRI, Hong Kong. Feb 2015 - Dec 2015.

Analyst Programmer, Hongkong Post, Hong Kong. Aug 2011 - Jan 2015.

Analyst Programmer, Prudential, Hong Kong. Dec 2010 - Aug 2011.

Consultant, Timeless Software Limited, Hong Kong. Sep 2004 - Nov 2010.