FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Effective Error Explanation Techniques for Concurrent Software

DISSERTATION

zur Erlangung des akademischen Grades

Doktorin der Technischen Wissenschaften

eingereicht von

Mitra Tabaei Befrouei, MSc.
Matrikelnummer 1229712

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Assistant Prof. Dipl.-Ing. Georg Weissenbacher, D.Phil.

Diese Dissertation haben begutachtet:

Georg Weissenbacher, D.Phil.	Prof. Dr. Rupak Majumdar	Prof. Dr. Thomas Eiter

Wien, 6. Oktober 2016

Mitra Tabaei Befrouei

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Effective Error Explanation Techniques for Concurrent Software

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktorin der Technischen Wissenschaften

by

Mitra Tabaei Befrouei, MSc.
Registration Number 1229712

to the Faculty of Informatics
at the TU Wien
Advisor: Assistant Prof. Dipl.-Ing. Georg Weissenbacher, D.Phil.

The dissertation has been reviewed by:

Georg Weissenbacher, D.Phil.	Prof. Dr. Rupak Majumdar	Prof. Dr. Thomas Eiter

Vienna, 6th October, 2016

Mitra Tabaei Befrouei

Erklärung zur Verfassung der Arbeit

Mitra Tabaei Befrouei, MSc.
Landstraßer Hauptstraße 45/1/17, 1030 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 6. Oktober 2016

Mitra Tabaei Befrouei

Acknowledgements

I would like to express my gratitude to those who paved the way for me during my PhD journey and helped me to pursue research by constantly giving support, advice, encouragement and inspiration. First and especially, I am very grateful to my advisor Dr. Georg Weissenbacher for his valuable advice, insightful comments and constant support. Working under his supervision was a rewarding and unforgettable experience. From him, I received ample freedom to find my own way and to decide on my own. Discussing and sharing ideas with him always inspired me. Especially, his vision and invaluable comments always triggered new ideas in my mind. Despite his busy schedule, he remained always accessible and happy to give support and advice in both professional and non-professional matters.

Looking further back, I would like to thank Prof. Dr. Stefan Leue from the University of Konstanz who led me during the first two years of my PhD. Among other things, I learned from him how to be precise while giving arguments and how to write a good research paper. I am honored that Prof. Helmut Veith (R.I.P.) interviewed me and gave me the opportunity to join his group, FORSYTE, at TU Wien. A special appreciation goes to Prof. Dr. Rupak Majumdar and Prof. Dr. Thomas Eiter who kindly accepted to review my dissertation as the examiners.

I would like also to thank my collaborators Dr. Thomas Wies and Dr. Daniel Schwartz-Narbonne from New York University and Dr. Chao Wang from Virginia Tech. Particularly, I must thank my amazing collaborator and former colleague Dr. Andreas Holzer for everything that I learned from him and for the great time that I had working with him.

I thank all my FORSYTE colleagues at TU Wien, especially Yulia Demyanova for being a great office mate and above all a great friend. I have been lucky to have the support and friendship of Yulia and her husband Dima. It has always been fun to have lunch breaks with them, and to chat about Russian and Persian traditions, customs, food, and the common words in our languages. I would like also to especially thank my colleagues Moritz Sinn for translating the abstract of my dissertation to German, Katarina Singer for doing the final corrections and checks on the resulting German translation, and Adrian Rebola Pardo for his very useful and meticulous comments on the beginning parts of my dissertation.

I thank the amazing Juliane Auerböck, Eva Nedoma, and Toni Pisjak for always treating me with a warm smile and providing assistance in administrative and technical matters.

I thank my friends in Vienna for bringing joy and fun to my PhD life, especially Serwah Sabetghadam my lively, lovely Iranian friend. It has been quite enjoyable and relaxing having a cup of tea with her, chatting about any subject from PhD thesis to our favorite TV serial, and at the end opening Hafez and reading and interpreting poems. I will never forget happy moments that we had in Vienna at concerts, hikes, parties, dancing courses, and at Uni.

Finally, I express my deep and sincere gratitude to my parents and my brother Ali for their unconditional love, encouragement and support. They have never stopped believing in me and encouraging me to pursue my dreams and fulfill my potentials.

To my very dear parents.

Kurzfassung

Fehler in nebenläufigen Programmen sind oftmals sehr schwierig aufzuspüren und zu erklären. Ein Hauptgrund hierfür ist, dass es für Menschen grundsätzlich schwierig ist, nebenläufige Programmausführungen zu verstehen und so die möglichen Verschränkungen der Ausführungsstränge vorauszusehen. Es kommt erschwerend hinzu, dass in verschiedenen Ausführungen eines nebenläufigen Programms die Interaktion der Ausführungsstränge und die Reihenfolge von Ereignissen unterschiedlich sein kann. Ein solches nicht-deterministisches Verhalten kann zu einem fehlerhaften und unerwünschten Verhalten des Programms führen, dessen Ursprung schwer zu analysieren ist. Die vorliegende Dissertation stellt effektive Techniken vor, welche den Programmierer oder die Programmiererin darin unterstützen, die Ursache von Fehlern in nebenläufigen Programmen zu ermitteln und zu verstehen. Unsere Techniken sind hierbei nicht auf bestimmte Fehlerarten beschränkt. Wir präsentieren ein allgemeines Framework, um Fehler sowohl in nebenläufigen Programmen, in denen die Ausführungsstränge über einen gemeinsamen Speicher kommunizieren ("shared memory multithreaded programs"), als auch in nebenläufigen Sytemen, in welchen die Kommunikation über den Austausch von Nachrichten stattfindet ("message passing concurrent systems"), zu finden.

Wir stellen Techniken basierend auf zwei unterschiedliche Ansätzen vor. Der erste Ansatz erkennt Anomalien in der Programmausführung mittels statistischer Analyse und der Extraktion und Auswertung von Mustern ("pattern mining"). Wir adaptieren einen weit verbreiteten Ansatz, das sogenannte "sequential pattern mining", um Anomalien aus Datensätzen bestehend aus fehlerhaften und korrekten Ausführungspfaden zu extrahieren. Anomalien in Form von Ereignisssequenzen legen die problematische oder unerwartete Reihenfolge nebenläufiger Ereignisse offen, welche Fehler verursachen können. Um die Skalierbarkeit des "sequential pattern mining"–Algorithmus sicherzustellen, schlagen wir drei verschiedene Approximationsmethoden vor, welche für jeweils verschiedene Problemstellungen geeignet sind. Die erste Methode unterapproximiert Anomalien, indem sie diese auf Sequenzen von Ereignissen beschränkt, welche in Ausführungspfaden hintereinander auftauchen. Die zweite Methode macht das Problem der Musterauswertung ("pattern mining") handhabbarer, indem Ausführungspfade mittels Partitionierung gekürzt werden. Die dritte Methode basiert auf einer neuen Abstraktionstechnik, welche die Skalierbarkeit des "sequential pattern mining"–Algorithmus erhöht. Diese Methode reduziert die Länge der Ausführungspfade und die Anzahl verschiedener Ereignisse in den Ausführungspfaden, bewahrt jedoch die Reihenfolge zwischen den Ereignissen einschließlich der Kontext-

wechsel. Kontextwechsel sind häufig entscheidend für das Verständnis von Fehlern in nebenläufigen Programmen. Die Fehlermuster, welche durch diese Methode extrahiert werden, legen nicht nur die problematischen Verschränkungen offen, sondern auch den Kontext in welchem der Fehler auftrat.

Unser zweiter Ansatz verwendet eine beweisbasierte Methode, um fehlerrelevante Abschnitte ("slices") fehlerhafter Ausführungspfade zu erkennen. Diese Technik analysiert einen einzelnen symbolischen Ausführungspfad, um ein Codefragment und einen Ablauf zu isolieren, welcher einen Fehler verursacht, und generiert außerdem Annotationen, die fehlerhafte Programmzustände beschreiben. Diese Technik erweitert existierende Techniken, die auf sequenzielle Programme beschränkt sind, für die Anwendung auf nebenläufige Programme.

Wir evaluieren die Effizienz und Effektivität unserer Techniken auf Benchmarks, welche eine große Bandbreite an Fehlern abdecken, wie sie in realen, nebenläufigen Programmen typischerweise vorkommen. Des Weiteren vergleichen wir die Stärken und Schwächen des Ansatzes basierend auf dem Auffinden von Anomalien mit jenen des beweisbasierten Ansatzes.

Abstract

Concurrency bugs are among the most difficult software bugs to detect and diagnose. This is mainly due to the inherent inability of humans to comprehend concurrently executing computations and to foresee the possible interleavings that they can entail. In concurrent programs, interactions between concurrent computations and the order of program events can vary across executions. This nondeterminism may result in erroneous and undesired program behavior whose root cause is difficult to analyze. Facing the challenge of debugging concurrent programs, this dissertation proposes effective concurrency bug explanation techniques to assist programmers in understanding the cause of failure in concurrent programs. Our techniques which do not rely on any characteristics specific to one type of bug, provide general frameworks for explaining bugs both in shared memory multithreaded programs and message passing concurrent systems.

In devising our dynamic techniques for bug explanation, we follow two different approaches, namely anomaly detection and slicing. Our anomaly detection techniques are based on statistical analysis and pattern mining. These techniques adapt a standard pattern mining algorithm called sequential pattern mining to extract anomalies from datasets of failing and passing execution traces. Anomalies in the form of sequences of events reveal the problematic or unexpected order between concurrent events which may cause a failure. To address scalability issues of the standard sequential pattern mining algorithm in extracting anomalies, we propose three different approximation methods according to the problem setting. The first technique under-approximates anomalies by limiting them to sequences of events which occur consecutively in traces. The second technique makes the pattern mining problem more tractable by shortening the length of the traces via partitioning them into subtraces. The third technique is based on a novel abstraction technique for improving the scalability of the pattern mining algorithm. This technique reduces the length and the number of distinct events in traces while preserving the ordering information between the events of original traces including context switches which are crucial for understanding concurrency bugs. The bug patterns extracted by this technique not only reveal the problematic interleavings but also the context in which the bug occurred.

In a completely different approach, we use a proof-based technique to construct semantics-aware slices from failing traces. This technique analyzes a single symbolic execution trace to isolate a slice comprising a code fragment and schedule causing a concurrency bug as

well as annotations that describe the erroneous program states. Our slicing technique, in fact, generalizes existing interpolation-based frameworks for sequential bug explanation to a concurrency setting.

We evaluate the efficiency and effectiveness of our proposed techniques on benchmarks covering a broad range of real-world concurrency bugs. Moreover, we compare the strengths and limitations of the anomaly detection techniques with the proof-based slicing technique.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Nowadays, parallel and concurrent systems ranging from multicore systems containing several cores on a single chip to Internet cloud systems spanning several different machines have become ubiquitous. Physical limitations hinder the performance of sequential computation to be increased according to the scaling trend projected by Moore's law. Consequently, system designers have considered parallelism as the primary means to improve the performance of computing systems.

To exploit the parallelism of hardware, programs and algorithms have been parallelized as well, and deployed in all kinds of computing systems–from Internet cloud systems to desktops to mobile systems. They are even used for controlling safety critical systems such as medical devices [LT93] and cars [KOESH07]. As a result, our modern life increasingly depends upon the correct and reliable functioning of concurrent systems.

In general, making the software correct and guaranteeing that it behaves according to its specification is difficult. The main barrier to software reliability is the large number of program behaviors which the programmer needs to consider in order to ensure that the program satisfies its specification. Due to challenges involved in making the software correct, it typically contains *bug*s, which are defects in the program code that can result in the failure. Failure is the failing or incorrect program behavior. Previous studies have shown that software bugs caused about 25–35% of system down time [MS00] and 50% of security vulnerabilities [CER].

Concurrency makes the matters even worse. A *concurrent program* comprises two or more pieces of sequential code that can be executed concurrently or simultaneously. In concurrent programs, the number of program behaviors increases exponentially with the number of concurrent components. Especially, interactions between concurrent computations significantly impact the number of potential program behaviors. Programmers

tend to think sequentially, therefore, they may easily overlook some of the interactions that their programs encompass. Unforeseen interactions between concurrently executing components may then result in erroneous and *nondeterministic* program behavior whose root cause is difficult to analyze. *Concurrency bugs* due to not properly synchronized concurrent components, therefore, widely exist in concurrent programs. A survey conducted at Microsoft in 2007 on 684 technical staffs revealed the frequency of this type of bugs: over 60% of respondents routinely have to deal with concurrency related bugs [GN08]. Moreover, bugs in concurrent systems can result in financial loss, severe injuries, or even the loss of life as history reminds us. In 1980s, a concurrency bug in Terac 25, an X-ray machine caused radiation overdoses which killed some patients and severely injured others [LT93]. In 2003, a race condition triggered in an energy management system caused power outage for two days in the northeastern America which resulted in around 6 billion dollars of financial loss [Pou04]. Recently, Nasdaq's Facebook IPO was delayed for half an hour due to a race condition which caused millions of dollars loss [Nas].

To improve software quality, *testing* and *model checking* have been primarily used. Testing tries to find bugs in a program by exploring its execution paths. Covering all execution paths is impractical, therefore testing techniques often guarantee *coverage* only according to some criteria, for instance, covering branches or statements. Testing is quite efficient in finding bugs and widely used in practice. However, it is not complete and it does not guarantee that the failure won't occur after deployment. On the other hand, model checking aims at providing a formal proof that the system is correct with respect to a given specification. If the system does not satisfy the specification, model checkers typically provide counterexamples as witnesses. However, model checkers scale poorly and their usage is limited in practice.

Testing and model checking of concurrent programs have been studied for years in academia and industry. The testing techniques for this type of programs mainly explore the space of program behaviors systematically or with random strategies to identify problematic interactions that cause the failure [MQ07, MQ06, Sen08]. These techniques provide various coverage guarantees over the space of interleavings.

Both testing and model checking only expose the bug by revealing the failing behavior of the program. They do not provide the root cause information for the revealed failure. *Debugging* is the process of understanding the cause of failures, locating and fixing bugs once their existence have been established. This process usually begins with *reproducing* the failing behavior, and subsequently *isolating* and *fixing* the code fragments responsible for the failure, using the observed failing behavior and the source code. In practice, programmers typically locate the bugs in their code using a highly involved, manual process. The debugging process which involves inspecting lengthy failing behaviors is, in general, notoriously time-consuming. A 2013 Cambridge University study found that software developers spend 50% of their programming time "fixing bugs" or "making code work" [BJGC13].

Debugging concurrent programs is even more challenging and time-consuming. This is because concurrency bugs are considered as *"Heisenbugs"* which rarely surface and are

hard to reproduce. Successive executions of a concurrent program do not necessarily produce the same results because of nondeterminism. This makes reproducing the failing behavior hard. In the Microsoft survey cited above, on average, developers spend seven days between finding a concurrency bug and applying a fix. In some cases, this may last up to several months. According to [JPPS11], Mozilla developers spent nearly two months to completely understand and fix a concurrency bug, even though they knew the specific interaction between the threads leading to the failure.

To debug concurrent programs, various concurrency bug detectors have been proposed. Bug detectors analyze source code or execution to isolate the concurrency bugs existing in the program. These detectors have in common that they focus on detecting *only* one particular type of bug such as data races or atomicity violations (see Chapter 2), hence relying on characteristics specific to that type of bug [LPSZ08]. Moreover, many of them have high false positive rates [EMBO10], therefore programmers may require more context to determine whether the results show the real bug.

Recently, more general frameworks have been proposed for debugging concurrent programs [LC09, LWC11, PVH10, PVH12, JTLL10, WWY+14, KKW15, GHR+15]. These techniques try to isolate the cause of failure once a failing behavior has been exposed via testing or model checking. However, some of them rely on given pattern templates for each type of bug [PVH10, PVH12, WWY+14], some may miss bugs due to not considering all the relevant ordering information [LC09, PVH10], and logical constraint based methods face scalability issues in practice [KKW15].

Considering the challenges involved in debugging concurrent programs and limitations of the current methods, there is still a great need for general frameworks to effectively address various types of concurrency bugs. This dissertation focuses on developing techniques to facilitate debugging concurrent programs. Our techniques do not exploit characteristics specific to one type of bug, thus addressing different types of bugs both in shared memory multithreaded programs and message passing concurrent systems. In addition, they do not rely on any given pattern templates or annotations.

In the remainder of this chapter, we first introduce the problem that we focus on in this dissertation which is explaining concurrency bugs or understanding the cause of failure in concurrent systems. This is done by defining basic notions of fault, error, failure and the constituent steps of software debugging process. We then briefly discuss the limitations of the state-of-the-art approaches in addressing this problem (see Chapter 2 for a detailed discussion). Finally, by describing the contributions of this dissertation in brief, we provide an overview of the approaches that this dissertation presents for addressing the problem.

1.2 Overview

1.2.1 Software Bugs

A *fault* is a defect in the program code which is also referred to as a *bug* in the software engineering literature. It is created at design or implementation time which makes the program not comply with the specification. It can also be created due to imprecise and incomplete specification of the program's functionality. In this dissertation, the terms bug and fault are used interchangeably, and our assumption is that the specification is precise and complete and bugs are due to the programmer's mistake at implementation time. As an example, *computation* faults are those faults in which a variable after a computation contains a value different from what was expected according to the specification. In C/C++, casts between signed to unsigned numbers can result in a computation fault because if the value of the signed primitive can not be represented using an unsigned primitive, it can produce an unexpected value. In Listing 1.1, if the error condition in the code above is met, then the return value of readdata() will be 4,294,967,295 on a system that uses 32-bit integers. If this value is passed to the standard memory copy or allocation functions, it may lead to an exploitable buffer overflow or underflow condition. Therefore, in this example cast between signed to unsigned at Line 6 is a fault which may cause a failure. As another example, in Listing 1.2, the function computes $|x1 - x2|$, however, the condition at Line 4 is wrong and should have been $x1 > x2$ instead. Due to the *faulty* condition of Line 4, in cases where $x1 \neq x2$ the assertion at Line 13 gets violated.

```
1 unsigned int readdata ()
2 {
3   int amount = 0;
4   ...
5   if ( result == ERROR )
6     amount = -1;
7   ...
8   return amount;
9 }
```

Listing 1.1: Signed to unsigned conversion fault

```
1 compute_diff (x1, x2)
2 {
3   if ( x1 != x2 ) {
4     if ( x1 < x2 )
5       diff = x1 - x2;
6     else
7       diff = x2 - x1;
8   }
9   else {
10    diff = 0;
11    ...
12  }
13  assert ( diff >= 0 );
14 }
```

Listing 1.2: Fault in a condition

During the execution of the program code, the fault may be *triggered* which means that the faulty code may be executed under certain conditions. When the fault is triggered, it creates an infection in the program state which is a discrepancy between the program state and the intended program behavior. We refer to an infected state as an *error*.

Every program state consists of values of program variables as well as the program counter

containing the current control location. The intended program behavior is defined by the program specification. An automatic way of comparing program states with the intended program behavior is by using the *assertion technique*. For example, in Listing 1.2, with the input (0,1) the program goes through the lines 3-5 and 13. After the execution of Line 5, the state is an *error* since $\mathsf{diff} = -1$ and according to program specification diff should always be greater than or equal to zero: $\mathsf{diff} \geq 0$.

A fault in the program code has to be triggered in order to cause an error, otherwise it does not necessarily leads to an error. For example, in Listing 1.2, with the input (2,2) the faulty statement at Line 4 gets never executed, consequently no error occurs during the program execution. If the error *propagates* during the remaining program execution, it leads to a *failure* which is the violation of the program specification. A failure is an observable error in the program behavior, for instance, a crash or an incorrect output. In Listing 1.2, with the input (0,1) the error occurring after the execution of Line 5 propagates to Line 13 in which the assertion gets violated. The violated assertion is an example of a failure.

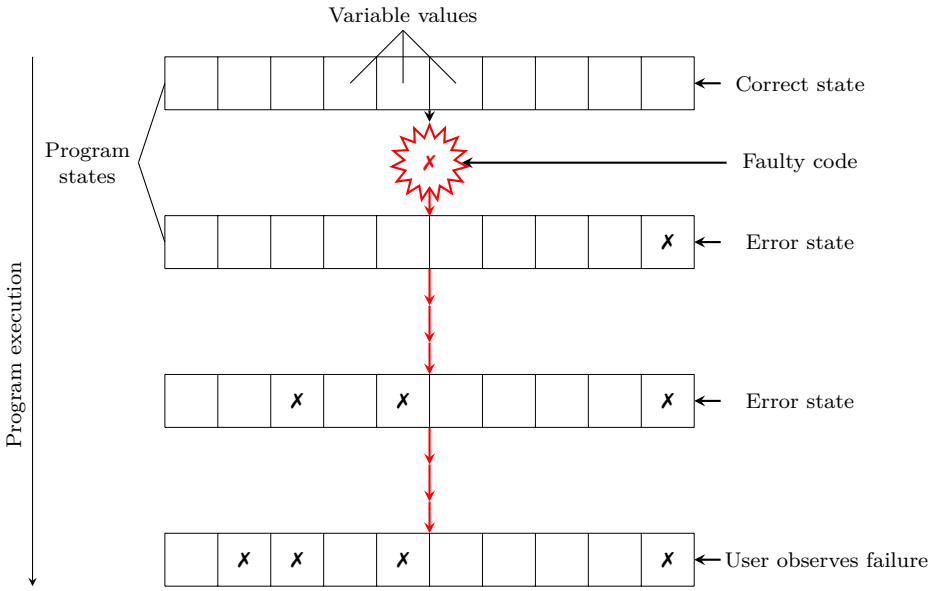


Figure 1.1: Propagation of error from fault to failure

A fault may not be propagated continuously, it may be overwritten, masked or corrected by some program action. Therefore, for the occurrence of a failure the fault in the code has to be triggered and then the error caused by the defect needs to be propagated. Every failure is thus caused by some error, and every error is caused by some earlier error originating at the fault which constitutes a cause-effect chain from fault to failure. Figure 1.1 illustrates the cause-effect chain from fault to failure [Zel09].

1.2.2 Debugging

When a program fails we want to realize the reason and fix it. *Debugging* is a software engineering process which starts after a failing behavior has been observed in a program, and can be decomposed into the following essential steps [Zel09, Chapter 1]:

1. Reproducing the failing behavior.
2. Analyzing the cause of failure by isolating the cause-effect chain from fault to failure.
3. Localizing the faulty part of the program code.
4. Fixing the fault.

We refer to the second step as *fault explanation* and the third step as *fault localization*. Among all the four steps of debugging, fault explanation is concerned with "understanding how the failure occurs", and is considered as the most challenging and time-consuming part of debugging. Fault explanation results in isolating the cause of failure and can greatly facilitate localizing and fixing the fault (third and forth steps). Reproducing the failing behavior (first step) can only be difficult for a nondeterministic program such as a concurrent program where the programmer does not have full control over the execution environment (we will introduce methods proposed for reproducing the behavior of concurrent programs in Chapter 2, Section 2.2.3).

Considering Figure 1.1, in order to find the fault we need to isolate the transition from a correct state to the error state. This process is both a search in *space* (we have to find out which part of the state is infected) and in *time* (we have to find out when the infection or error takes place). These searches in space and time are quite challenging even for simple programs. Every state can contain up to millions of variables. Moreover, a program execution can have several thousands or up to millions of these states. Therefore, space and time form a wide area in which only two points are known: the initial state which is entirely correct and the final failure state which has an infected part. Within the area spanned by space and time, the aim of debugging is to locate the fault which is a single transition from a correct state to an error state which eventually causes the failure. Figure 1.2 illustrates this process [Zel09, Chapter 1].

In principle, all debugging problems can be solved manually by analyzing source code and observing the failing program behavior at run time. In other words, debugging can be done by having source code and a failing execution trace. As discussed previously, this process which involves search in time and space is quite time-consuming and tedious. Therefore, a large number of automated debugging techniques have been proposed to reduce the manual effort involved in debugging. Specially for debugging sequential programs, a wide variety of tools and techniques have been proposed [WD09, Ali12]. According to the underlying principles and assumptions, we classify these approaches into three broad categories, namely *slicing*, *anomaly detection* and *causality analysis*.

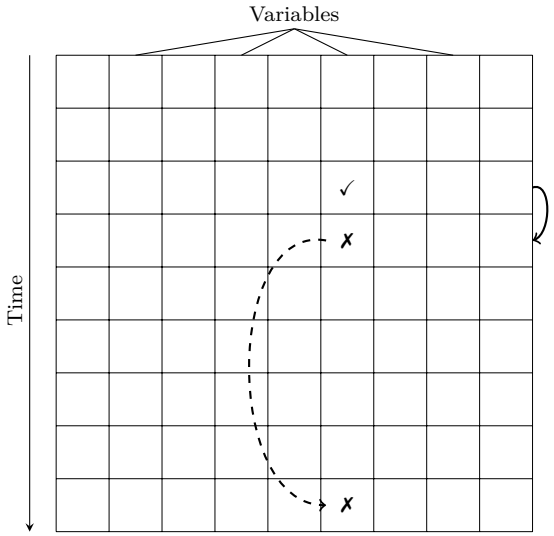


Figure 1.2: Isolating the fault: search in space and time. A fault manifests itself as a transition from a correct state (✓) to an error state (✗), where a faulty statement causes the initial infection.

- Slicing techniques: They effectively *narrow* down the search for the root cause of a failure by identifying the *relevant* statements and variables to the failure. Therefore, to locate the faulty statements the programmer needs only to focus on the slice.
- Anomaly detection techniques: They extract *anomalies* by examining differences between passing and failing execution traces. Anomalies are properties that are only common or frequent in failing traces but not in passing traces. Similar to slices, anomalies can effectively *narrow down* the search for the root cause of the failure.
- Causality analysis techniques: They aim at locating the fault or the *root cause* of the failure by isolating the cause-effect chain from the fault to the failure in a failing trace.

We present a detailed discussion of each of these categories in Chapter 2, Section 2.1. As we will see, the techniques belonging to one of the categories introduced above have been mainly proposed for debugging sequential programs, and only in a few cases for debugging concurrent programs. Since the techniques proposed for exposing and detecting concurrency bugs are based on the specific characteristics of concurrency bugs, we treat them as a separate category.

In the following, we define the basic notions of *concurrent programs* and *interleaving semantics*. We then provide a brief introduction of the common types of *concurrency bugs* and how they have been addressed in the previous work. A detailed discussion of concurrency bugs and concurrency bug detectors is given in Chapter 2, Section 2.2.1.

1.2.3 Concurrent Programs and Interleaving Semantics

A *sequential program* specifies sequential execution of a list of statements. Therefore, it imposes a *total ordering* on the execution of the statements it specifies which results in a *single* thread of control. A *concurrent program* comprises two or more pieces of sequential code that may be executed concurrently or simultaneously. Unlike sequential programs, a concurrent program has multiple threads of control in which every *thread* corresponds to the execution of one of the pieces of sequential code. Therefore, each thread specifies a total order on the execution of its statements. Since by default there is no *order* between the execution of the statements of different threads, a concurrent program imposes a *partial* ordering on its statements. As a consequence, when a concurrent program is executed repeatedly even with the same input the actions from different threads may be executed in different orderings. We refer to the ordering of execution of actions in a concurrent program as *scheduling*. Since the scheduling of a concurrent program may vary every time it is executed, it may reveal different behaviors even with the same input. This property of concurrent programs is referred to as *nondeterminism*.

Interleaving semantics is widely used for modeling the nondeterministic behavior of concurrent programs. We assume that the statements (actions) of a thread are *atomic*, which means that their execution is not interrupted and they are always executed to the completion. Two atomic actions a and b from two different threads are *concurrent*, denoted by $a \| b$, if either a can be executed before b or vice versa. Using interleaving semantics we can model executions of concurrent programs as total orders by nondeterministically choosing one order between concurrent actions.

In a concurrent program, multiple threads interact via *communication* and *synchronization*. Communication mechanisms for concurrent programs are generally based on either *shared memory* or *message passing*. In the shared memory mechanism, some or all of a program's variables are accessible to multiple threads. These variables are referred to as *shared variables*. Communication between the threads then happens via read and write accesses to shared variables. In the message passing mechanism, there is no shared variable. For communication, threads perform explicit *send* and *receive* operations to transmit data via message passing channels.

As we discussed, in a multithreaded program by default there is no order between the execution of the statements of different threads. Synchronization is a mechanism for imposing order on the concurrent actions of different threads. In message passing systems, synchronization is generally implicit: a message must be sent before it can be received. If a thread attempts to receive a message that has not yet been sent, it will wait for the sender to catch up. However, in shared memory systems synchronization is not generally implicit. Language constructs such as *locks*, *monitors* and *semaphores* are used for synchronizing accesses to shared variables, otherwise, a receiving thread could read the old value of a variable, before it has been written by the sender.

During the execution of a concurrent program, actions of each thread are executed in the program order of that thread and synchronization operations determine the order between

the actions of different threads. Since a priori there is no order between the synchronization points of different threads, timing of operations or *scheduler* determines the order in which threads reach synchronization points. The variation in the order in which threads arrive at synchronization points can in turn change the order of other operations in the execution. Therefore, an execution follows a schedule nondeterministically. As we have already discussed, different schedules can result in different outcomes even with the same input.

1.2.4 Concurrency Bugs - Concurrency Bug Detectors

Due to nondeterminism, communication and synchronization are two main challenges in designing and implementing concurrent programs, and consequently the main sources of *concurrency bugs*. Concurrency bugs can lead to unintended program behavior as the result of particular *ordering* of actions in different threads. In fact, the failures caused by concurrency bugs depend on the execution schedule. In other words, concurrency bugs manifest only in a particular execution schedule. Figure 1.3 shows a typical concurrency bug due to scheduling. In this figure, Thread 1 stores a value into a pointer variable called pRec and then dereferences it later. If Thread 2 nullifies pRec (statement 3) before Thread 1 dereferences it (statement 2), the program crashes. However, if statement 3 is executed after statement 2, the program does not crash. Since in practice particular schedules which lead to a failure occur rarely, detecting and diagnosing concurrency bugs is quite challenging.

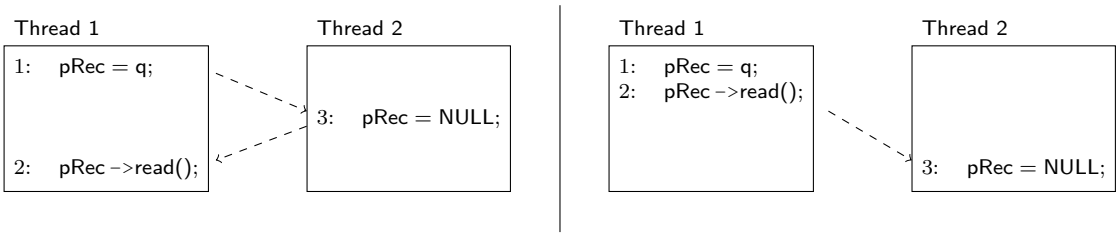


Figure 1.3: A typical concurrency bug due to scheduling: The program crashes with the scheduling $1 \rightarrow 3 \rightarrow 2$ (left) but not with the scheduling $1 \rightarrow 2 \rightarrow 3$ (rigth).

To detect and explain concurrency bugs, researchers have focused on common types of concurrency bugs [LPSZ08] such as data races (concurrent conflicting accesses to the same memory location) and atomicity/serializability violations (an interference between supposedly indivisible critical regions). The detection of data races requires no knowledge of the program semantics and has therefore received ample attention (see Chapter 2). Freedom from data races, however, is neither a necessary nor a sufficient property to establish the correctness of a concurrent program: benign data-races include races that affect the program outcome in a manner acceptable to the programmer [EMBO10]. In particular, it does not guarantee the absence of atomicity violations, which constitute the predominant class of non-deadlock concurrency bugs [LPSZ08]. Atomicity violations

are inherently tied to the intended granularity of code segments (or operations) of a program. Automated atomicity checking therefore depends on heuristics [XBH05] or atomicity annotations [FQ03] to obtain the boundaries of operations and data objects.

The past two decades have seen numerous tools for the exposure and detection of data races [SBN+97a, NM91b, EA03b, FF10, EQT10], atomicity or serializability violations [FQ03, LTQZ06, XBH05, PLZ09], or more general frameworks which also address order violations (execution of threads in an unintended order) [LC09, PVH12]. As we will discuss in Chapter 2, the majority of these techniques address only one specific type of concurrency bugs, thus lack generality. Moreover, they rely on given heuristics or bug patterns which restrict their applicability.

In this dissertation, we propose formal proof-based as well as statistical methods for concurrency bug explanation. Our techniques do not exploit any characteristics specific to one type of concurrency bug, therefore providing general frameworks for concurrency bug explanation. We consider both shared memory multithreaded programs and message passing concurrent programs. Our *statistical* techniques for concurrency bug explanation are based on anomaly detection and our *proof-based* technique is based on slicing.

1.3 Contributions

This dissertation makes contributions to the field of debugging concurrent programs by proposing general frameworks for concurrency bug explanation. Empirical results obtained by the implementation of these frameworks show the effectiveness as well as efficiency of the proposed techniques in explaining concurrency bugs. We divide our techniques into two categories of anomaly detection and slicing.

1.3.1 Anomaly Detection for Debugging Concurrent Systems

Our anomaly detection techniques are based on mining datasets of execution traces. We use a standard pattern mining method called *sequential pattern mining* for extracting anomalies from datasets of failing and passing execution traces. Anomalies in the form of sequences of events reveal problematic interleavings or interactions in a concurrent system. With these techniques, we address bugs both in multithreaded shared memory programs and concurrent systems with message passing mechanism.

Message Passing Concurrent Systems. We present techniques for explaining *counterexamples* indicating the violation of a desired property in a message passing concurrent system. The counterexamples are obtained by explicit state *model checking* of a faulty concurrent system. In these techniques:

- We formalize the notion of *explanatory sequences* as the anomalies that are extracted through mining datasets of counterexamples and correct execution traces.

- We show the intractability of standard pattern mining algorithms in extracting explanatory sequences due to the combinatorial explosion of the potential candidates.
- To address the scalability issues in applying standard pattern mining algorithms for extracting explanatory sequences, we propose two approximation techniques.
- We evaluate the efficiency and effectiveness of the proposed approximations in practice.

Shared Memory Multithreaded Programs. Our anomaly detection technique for understanding bugs in shared memory multithreaded programs is based on mining execution traces produced as the result of *testing* these types of programs. In this technique:

- We formalize the notion of *bug explanation pattern* as the anomalies that are extracted through mining datasets of failing and passing execution traces. We provide also the underlying theoretical rational.
- We propose a novel *abstraction* technique for making the problem of mining bug explanation patterns tractable.
- We show the effectiveness and efficiency of our abstraction technique in analyzing lengthy execution traces of real world applications.

1.3.2 Semantics-aware Slicing Technique for Debugging Concurrent Systems

While our anomaly detection approach is based on analyzing a set of concrete execution traces, in our slicing approach we analyze a single *symbolic execution*. In this technique, a failing trace is translated into an unsatisfiable logical formula. From a proof of unsatisfiability *interpolants* are extracted. These interpolants are then used to construct a slice of the failing trace that abstracts from the irrelevant statements and explains the faulty behavior.

Recent work [ESW12, CESW13, MSTC14] showed how *interpolation* can be used to construct semantics-aware slices in sequential software. In our slicing technique:

- We lift the existing formal framework of interpolation-based slicing to a concurrency setting:
  - We consider control- and data-dependency between threads in addition to intra-thread dependencies, adding the ability to reflect hazards such as race conditions and atomicity violations.
  - We prove that the slices produced using the hazard-aware interpolation are sound and sufficient to trigger the failure.

- We demonstrate the effectiveness of hazard-aware interpolation in explaining common types of concurrency bugs through the empirical evaluation.

1.3.3 Outline

Chapter 2 introduces state-of-the-art (semi)automated software debugging techniques. Especially, this chapter focuses on the three fundamental approaches proposed to automate debugging software, namely slicing, anomaly detection and causality analysis. We discuss state-of-the-art concurrency bug detectors. We provide also a discussion of the strengths and weaknesses of these approaches.

The remaining chapters present our contributions listed previously in this section. Chapters 3, 4 cover the anomaly detection methods for counterexample explanation in message passing concurrent systems. Chapter 5 presents the anomaly detection method for explaining bugs in multithreaded shared memory programs. Chapter 6 is dedicated to our interpolation-based slicing technique for concurrent programs. Finally, we provide some concluding remarks and a discussion of future work in Chapter 7.

The materials in Chapters 3-6 which cover our contributions listed in Sections 1.3.1 and 1.3.2 are published as journal or conference papers. The techniques in Chapters 3 and 4 are presented in [LTB12] and [LTB13], respectively. The abstraction and mining method of Chapter 5 is presented in [TBWW16] and [TBWW14]. [TBWW14] was nominated for the best paper award at the *Runtime Verification* conference 2014, and invited for a special edition of FMSD [TBWW16]. The interpolation-based slicing method of Chapter 6 is presented in [HSNTB$^+$16].

CHAPTER 2

Background and Previous Work

2.1 Automated Debugging Techniques

In Section 1.2.2, we introduced the three fundamental approaches proposed for automating and facilitating debugging software. These approaches, namely slicing, anomaly detection and causality analysis, are discussed in detail in this chapter. Furthermore, we provide a discussion of their strengths and limitations. Although these approaches have been proposed mostly for debugging sequential programs, they have also been applied in some cases for debugging concurrent programs.

In Section 2.2.1, we discuss the state-of-the-art concurrency debugging tools. We provide a more detailed discussion of these tools and techniques in comparison with our own techniques in the "Related Work" sections of the Chapters 3-6. In Section 2.2.3 of this chapter we present the related work which focuses on exposing and reproducing concurrency bugs.

2.1.1 Slicing

A *program slice* consists of statements of a program that (potentially) affect the values computed at some point of interest in the program, referred to as a *slicing criterion* [Tip95]. A slicing criterion usually consists of a pair (program location, variable). Weiser first introduced the concept of a program slice [Wei81], and considered slices as abstractions programmers make while doing debugging. To understand the cause of the failure, for every variable with an incorrect value in the failing state, we identify the statements or values which could have influenced it. These are the *relevant* statements which could have potentially caused the failure. By identifying relevant statements, at the same time, the statements which are *irrelevant* to the failure are also identified. Isolating the relevant statements helps us to effectively narrow down our search space for the cause of the failure by focusing on relevant values and statements and ignoring the irrelevant ones.

To isolate a program slice which is relevant to a failure, we need to reason backward starting from the failing state. For example, the piece of code in Listing 2.1 outputs the value of x which is always zero. If we reason backward from the statement "`cout << x;`" (Line 5), we find that the value of x in this statement comes from the assignment at Line 4. Similarly, the value zero of y at Line 4 comes from the assignment at Line 3. Therefore, only statements at Lines 3 and 4 are relevant for the value of x at Line 5. Moreover, the input value of x at Line 2 is irrelevant for its value at Line 5.

```
1 int x, y;
2 cin >> x;
3 y = 0;
4 x = y;
5 cout << x;
```

Listing 2.1: Example: relevant/irrelevant statements to a slicing criterion

Backward reasoning for determining relevant/irrelevant statements can be done on the source code without executing the program. However, for this reasoning we need to know the possible *order* of execution between the statements because in an execution only earlier statements may influence later statements. To this end, a *control flow graph* (CFG) [Muc97, §7] which is derived from the source code and shows the possible flow of control between the statements can be used. In a control flow graph, each statement of a program is mapped to a *node*. Edges between the nodes represent the possible flow of control between the statements. For example, an edge from the statement A to the statement B means that during execution B may be executed immediately after A. For example, Figure 2.1 depicts control flow graph of Listing 2.2.

For computing program slices in addition to knowing the order of execution of statements, we need to know how the statements affect each other. Statements can influence each other in two ways. Either one statement writes a value to a variable which is used by another statement or one statement control the execution of another statement. Therefore, there are two types of *dependencies* between the statements in a program:

- Data Dependency: statement B is *data dependent* on statement A, if:
  - A writes a variable like V which is read by B, and
  - there is at least one path in the control flow graph from A to B in which V is not written by any other statement.
- Control Dependency: statement B is *control dependent* on statement A, if outcome of A determines whether B is executed.

For example, in Listing 2.2 statement at Line 3 is data dependent on statement at Line 2 because it reads a value of x which is written at Line 2. Similarly, statements at Lines 4 and 6 are also data dependent on Line 2. Moreover, statements at Lines 4 and 6 are control dependent on statement at Line 3 because the outcome of "`x >= 0`" determines

which one will be executed. Figure 2.1 shows control flow graph of Listing 2.2 along with control- and data-dependencies between the nodes. A graph like Figure 2.1 is a *program-dependence graph* since it reflects all the dependencies within a program.

```
1  int x, y;
2  x = 10;
3  if (x >= 0)
4    y = x;
5  else
6    y = -x;
```

Listing 2.2: Example: data- and control-dependencies

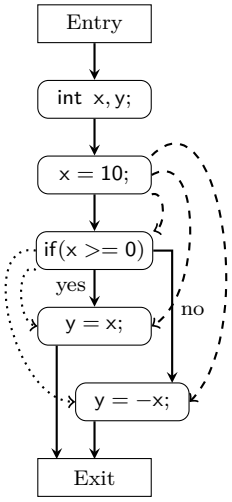


Figure 2.1: Listing 2.2 Dependence graph, control-dependencies: dotted arrows (left), data-dependencies: dashed arrows (right)

Using control- and data-dependencies, we can isolate a program slice which may have influenced a specific statement. The corresponding operation is called *slicing*. The slice consisting of a set of statements that can influence a given statement is referred to as the *backward slice* of that statement. Formally, the backward slice of the statement B, $S^B(B)$, is defined as $S^B(B) = \{A | A \rightarrow^* B\}$ in which $A \rightarrow B$ denotes a dependency relation: B is data- or control-dependent on A, and $*$ denotes the transitive closure of this dependency relation [Zel09, Chapter 7]. For example, the backward slice of Line 4 in Listing 2.2 consists of Lines 2 and 3. Similarly, the *forward slice* of a given statement A is defined as the set of all statements that can be influenced by A.

Static Slicing

When a slice is computed from the source code and by using control- and data-dependencies, it is referred to as a *static* slice since only statically available information is used for computing the slice. Static information is obtained by analyzing the source code without executing the program. There exists a large body of work on static slicing of programs. The original definition of program slicing which first introduced by Weiser in [Wei84] is based on iterative solution of dataflow equations [Tip95]. For every node of control flow graph (CFG), dataflow equations are defined by taking data-dependencies into account. These equations are then solved in an iterative process until a fixed point is reached. Ottenstein and Ottenstein defined slicing as a reachability problem in the dependence graph of a program [OO84]. As we have seen, PDG or program dependence graph

represents the data- and control-dependencies between the statements of a program. Various algorithms have also been proposed for static slicing of programs in the presence of unstructured control flow (statements such as goto, break, continue), composite data types and pointers, interprocedural static slicing, and static slicing of distributed programs. Detailed discussion of these algorithms can be found in [Tip95].

As we have already discussed, slicing reduces the debugging search domain based on the idea that the cause for an incorrect variable value at a statement can be found in the static slice associated with that variable-statement pair. In fact, slices are typically much smaller than original programs, therefore using them greatly speed up the debugging process. Binkley and Harman [BH03] examined size of slices in 43 C programs and found that on the average a backward slice contains under 30% of the original programs.

However, static analysis methods for computing the slices from the source code are imprecise or unsound. For example, Findbugs [1] which is a static analyzer tool has a false positive rate of 50% [Zel09, Chapter 7]. Producing false positives is due to the fact that, in general, most interesting problems in static analysis are undecidable. Consequently, static analysis techniques resort to conservative approximation. Arrays and pointers are examples of constructs that make tracking dependencies difficult.

Dynamic Slicing

Similar to a static slice, a dynamic slice encompasses a part of the program that could have influenced a specific statement. However, a dynamic slice is extracted from a single concrete run, so it does not hold for all possible runs of a program like a static slice. Moreover, it uses the dynamic information obtained from the program execution. Since debugging starts after observing a failing behavior, it makes more sense to construct a slice from the failing run than from the entire program for understanding the cause of the failure. Dynamic slices are more precise than static slices since they exploit runtime information. Moreover, they are much smaller compared to static slices. While a static backward slice typically encompasses 30% of a program code, a dynamic slice only encompasses 5% of the executed statements. Furthermore, the executed statements form a subset of all statements of the program [Zel09, Chapter 9].

To compute a dynamic slice, a *trace* which is an ordered list of statements during the program execution is required. Traces can be generated by *instrumenting* the program using tools like Cil[2], Llvm[3] or Pin[4]. In Figure 2.2, a sample trace is given which is generated while executing the code snippet in Listing 2.3 with input $\mathsf{n} = 2$. The trace shows the execution history, and is a sequence of events each corresponding to the execution of a statement in Listing 2.3. When a statement is executed more than once, the corresponding events are distinguished with indices. Following the data- and control-dependencies for the statement at Line 12 shows that the static slice of this statement

[1] http://findbugs.sourceforge.net/
[2] http://sourceforge.net/projects/cil/
[3] http://llvm.org/
[4] https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

contains the entire code. However, the dynamic slice of this statement computed from the trace in Figure 2.2 excludes the statement at Line 8. Note that for input $\mathsf{n} = 2$, the loop is executed twice and the assignments $\mathsf{x} = 18$ and $\mathsf{x} = 17$ are each executed once. Since the assignment of 18 to x in the first iteration of the loop is "killed" by the assignment of 17 to x in the second iteration, the statement at Line 8 is excluded from the dynamic slice.

```
1  int n, i, x, z;
2  cin >> n;
3  i = 1;
4  while (i <= n)  {
5    if (i \% 2 == 0)
6      x = 17;
7    else
8      x = 18;
9    z = x;
10   i++;
11 }
12 cout << z;
```

Listing 2.3: Example: signed to unsigned conversion fault

2^{1} cin >> n;
3^{2} i = 1;
4^{3} i <= n;
5^{4} i%2 == 0;
8^{5} x = 18;
9^{6} z = x;
10^{7} i++;
4^{8} i <= n;
5^{9} i%2 == 0;
6^{10} x = 17;
9^{11} z = x;
10^{12} i++;
4^{13} i <= n;
12^{14} cout << z;

Figure 2.2: A sample trace of Listing 2.3 with input $n = 2$

Different techniques have also been presented for dynamic slicing. A survey of dynamic program slicing can be found in [Tip95]. As we discussed, dynamic slicing is more suitable for debugging. However, it causes overhead since it requires logging of traces in order to record which statements are executed in which order. Moreover, it lacks generality: Dynamic slices are only applicable to a single run and cannot be generalized to other runs.

2.1.2 Anomaly Detection

Similar to slices, *anomalies* which are properties that are only common or frequent in failing runs but not in passing runs can effectively narrow down the search for faults in the code. Anomalies as common or frequent properties of "only" failing runs are just *abnormal* behaviors which are, in turn, a deviation from the normal behaviors. They are not incorrect behavior, therefore they do not imply a defect or fault. However, they can be good indicators of faults and they are considered as potential defects. In other words, abnormal properties of a program run are more likely to indicate defects than normal properties of the run. Therefore, in searching for faults it makes sense to first extract anomalies and then inspecting them for finding the faults. Anomalies are characterized by certain properties of the program run such as:

- *Code coverage*: Code that is (frequently) executed in failing runs but not in passing runs.
- *Call sequences*: Sequences of function calls that (frequently) occur only in failing runs.
- *Variable values*: Variables that (frequently) take certain values in failing runs only.

Techniques for extracting anomalies examine the differences between the common properties of a set of passing runs and a set of failing runs. One of the simplest methods for detecting anomalies is comparing the *code coverage* of passing and failing runs. To this end, a *coverage tool* which instruments code such that the execution keeps track of all lines being executed is required. These tools are usually used for assessing the quality of a test suite. Therefore such coverage information from testing can be visualized to guide the user in detecting anomalies. Tarantula is a tool for visualizing coverage anomalies [JHS02]. In this tool color is used to visually map the participation of each program statement in the outcome of the execution of the program with a test suite, consisting of both passed and failed test cases. Based on this visual mapping, a user can inspect the statements which are only or frequently executed in failing runs to locate potentially faulty statements. A case study in [JHS02] shows that focusing on abnormal statements allowed programmers to ignore 80% of the code.

Renieris and Reiss also proposed a coverage based fault localization method [RR03]. However, in their method, rather than comparing the coverage information of a set of passing runs with a set of failing runs, a single failing run is compared with its *nearest neighbor*. The nearest neighbor of a failing run is a passing run whose coverage is the most similar to the failing run as measured by a distance metric on coverage sets of program runs. Using these two runs, the method in [RR03] removes the set of statements executed by the passed run from the set of statements executed by the failed run(*set difference*). The resulting set of statements is the initial set of statements from which the programmer should start searching for the fault. In a case study, Renieris and Reiss [RR03] showed that nearest neighbor approach predicts the location of faults better than two other coverage based methods (these two methods are similar to those presented in [AHLW95] and [PS92]). In this work, in 17% of all test runs of the Siemens test suite [HFGO94], the location of the fault could be narrowed down to 10% or less of the program code.

In coverage based fault localization methods, single statements are considered to be correlated with failure. However, in some cases failures occur due to a *sequence* of method calls tied to a specific object. For example, using streams in Java without explicitly closing them can lead to failure. If too many files are left open before the garbage collector destroys the unused streams, file handles may run out which results in a failure. Therefore, a sequence of method calls to a stream which is not followed by a call to close indicates a fault. Dallmeier et al. [DLZ05] proposed a lightweight technique for detecting failure-correlated call sequences in a Java program. In this technique, the method call sequences from multiple passing runs and one failing run are compared. During runtime, for each object the *call-sequence sets* which contain short sequences (of some specific length like k) of consecutive calls initiated by the object are captured. Every class, is then

assigned a *weight* according to the comparison of the call-sequence sets of its objects from multiple passing runs and one failing run. Classes with higher weights exhibit many call sequences only present in the failing run. Therefore, they are considered more suspicious. In a case study, Dallmeier et al. [DLZ05] found that sequences of calls always predicted defects better than simply comparing coverage. Overall, the technique pinpointed the faulty class in 36% of all test runs with similar low cost as capturing and comparing code coverage.

The main problem with the approaches described above is that they heavily rely on the quality of the test suite. Therefore, the resulting set of suspicious statements that they produce depends on which failing and passing program runs have been used for comparison. In these methods, when the initial set of *suspicious* statements does not contain the fault, they provide a ranking on the remaining statements of the program based on their proximity to the suspicious statements in the *System Dependence Graph* or SDG [RR03]. A System Dependence Graph (SDG) models inter- and intra- dependencies both control and data between statements of a program while PDG models dependencies inside a procedure.

In addition to code coverage, there are other aspects of program runs that can be used for anomaly detection. Liblit et al. [LAZJ03, LNZ+05] proposed a fault localization technique which instruments programs with *predicates* at particular points and monitors their values during program execution. One type of predicates they use is for tracking the return value of functions. In C functions, the sign of the return value is often used to indicate success or failure. In their method, at each scalar returning function call site, they track six predicates showing whether the returned value is ever < 0, ≤ 0, > 0, ≥ 0, $= 0$, or $\neq 0$. Finally, predicates are pruned and ranked according to some statistical measures for inspection by the programmer. This method has been designed for isolating bugs in programs after they have been deployed. It uses the data sampled at instrumentation points while the program is being executed at users' machines.

We have seen different properties that can be collected from multiple runs for anomaly detection. Another approach for leveraging multiple runs is to generate likely *invariants* which hold for all runs and then use them for detecting anomalies. Invariants identify program properties which are valid for all program runs. Ernst et al. [ECGN01] proposed a dynamic invariant detection technique implemented in a tool called Daikon for discovering invariants that hold for all observed runs. In Daikon, invariants for methods (specification) are in the form of pre- and post-conditions. For the post-condition of shell_sort(), for instance, the invariant specifies that the return value of the method is the sorted form of the input array. Daikon maintains a library of patterns of invariants over variables and constants. A program is instrumented in a way that all values of all variables at all entries and exits of all functions are logged into a trace file during runtime. After program execution, the trace file is analyzed by Daikon to extract likely invariants based on the values of variables. Daikon's output is a set of likely invariants that are statistically justified by the trace file. The obvious drawback of Daikon is that it is limited to a library of patterns for extracting invariants. Therefore, it cannot detect

invariants whose patterns do not exist in its library. In general, new patterns can always be added into Daikon's invariant library. However, a large number of the patterns increases the running time of Daikon. For each program point, invariant detection time is cubic in the number of variables that are in scope at that point because patterns involve at most three variables.

Diduce [HL02] is another invariant inference tool which infers invariants on-the-fly specifically for detecting anomalies and isolating bugs. In contrast to Daikon, Diduce computes invariants while program is running. However, it infers only a very specific set of invariants which are defined on the values of a set of tracked expressions at various program points. During "training" mode, Diduce learns invariants by relaxing invariant hypotheses as needed during a run. In "checking" mode, it reports about invariant violations (or relaxations) which occur along the way. Although Diduce is more limited than Daikon in terms of inferring various forms of invariants, it has been shown to be effective in detecting bugs in a number of programs [HL02].

2.1.3 Causality Analysis

Slicing and anomaly detection can only effectively narrow the search for actual faults in the code. They isolate the potential faults but not the actual faults which caused the failure. However, causality analysis techniques aim at locating the actual faults. Debugging itself is, in fact, performing a *causality analysis*. This is due to the fact that in the search for a fault we are, actually, looking for the cause-effect relationship.

Philosophers have long been struggling with the problem of what it means for one event to cause another. Lewis has proposed a theory of causality in which a cause is something that makes a difference: If there had not been a cause c, there would had not been an effect e (*counterfactual dependence*) [Lew01]. In this theory, we have the notion of possible *worlds*, *alternative* worlds and similarity between these worlds which is evaluated based on the notion of distance metrics between the worlds. According to Lewis' counterfactual reasoning, an effect e is dependent on a cause c in a world w if and only if in all worlds most similar to w in which $\neg c$ it is also the case that $\neg e$. Lewis, then, equates causality to an evaluation on the basis of the distance metrics between possible worlds.

Automated debugging approaches aiming at isolating the actual cause mostly use Lewis' counterfactual reasoning as the causality framework. In these approaches, execution traces are considered as possible worlds. Effects are the observed failure which are typically a violated assertion or a crash. A passing trace which is the most *similar* to a failing trace is considered as an alternative world. Different techniques use different distance metrics to compute the similarity between the traces. Causes are, then, defined as the differences between a failing and the most similar passing trace.

Delta Debugging

Delta Debugging detects causes of failure by systematically minimizing the difference between a failing and a passing run through several experiments. It differs from program

analysis in that it is purely experimental, so it does not require knowledge or analysis of the program code. The authors of [ZH02] introduce delta debugging as a general fully automatic approach for debugging: "*Delta Debugging can be applied to all circumstances that in any way affect the program execution. Delta Debugging is fully automatic: whenever some regression test fails, an additional Delta Debugging run automatically determines the failure-inducing circumstances.*"

Delta Debugging requires as input a failing run $r_{\text{✗}}$ and a passing run $r_{\checkmark}$. The *difference* between these two runs is defined as δ so that applying δ to $r_{\checkmark}$ produces $r_{\text{✗}}$: $\delta(r_{\checkmark}) = r_{\text{✗}}$. For example, δ can represent the difference between inputs of $r_{\text{✗}}$ and $r_{\checkmark}$ [ZH02], the difference between two corresponding states of $r_{\text{✗}}$ and $r_{\checkmark}$ [Zel02], or in concurrent programs the difference between the thread schedules of $r_{\text{✗}}$ and $r_{\checkmark}$ [CZ02], as we will explain in the following. To find the cause of failure, δ is systematically minimized through several tests or iterations. To this end, the initial change δ is decomposed into a number of elementary changes $\delta_1, \ldots, \delta_n$. The method then searches for a *minimal* subset of elementary changes which causes the failure. The minimality of this subset is defined as removing any single change from it would cause the failure to disappear. The search is done via a modified binary search in which the initial set of elementary changes is partitioned into subsets during the iterations of algorithm. At every iteration, the chosen subset of changes is then applied to $r_{\checkmark}$ using a *test* function. The algorithm assumes the existence of a *test* function which applies a set of elementary changes to $r_{\checkmark}$ and determines the outcome which can be *failing*(✗), *passing*(✓) or *undefined*(?) [5]. In *best case*, the algorithm performs like a binary search while in worst case, the number of iterations of algorithm or test runs is quadratic in the number of elementary changes.

In [ZH02], Delta Debugging is used for simplifying and isolating failure inducing inputs. When a program fails given some input, not all the elements of input are relevant to the failure. For example, if the input is a C file or an HTML file, usually only a small part of it is causal or relevant to the failure. Therefore, in [ZH02], cause is defined in the *program input* and isolated by systematically narrowing down the difference δ between the actual input where the failure occurs and another given input where the failure does not occur.

Although simplifying the failure-inducing input facilitates the debugging process, it does not reveal the faulty parts of the code which need to be fixed. In addition, method of [ZH02] has some other limitations. In this method, decomposition of δ into elementary changes is problem specific and the method does not suggest any general way of decomposition. In case of a C-file as input, for instance, each δ_i can correspond to the i^{th} C token of the C-file. On the other hand, the performance of the algorithm heavily depends on the decomposition of δ, which can only be done by knowing the structure of the input. Moreover, the performance of the algorithm depends on how the set of elementary changes are partitioned into subsets during the search. The efficient

[5] The function *test* returns ✗ if the original failure occurs. To this end, *test* returns ✗ if the run crashes at the same location as $r_{\text{✗}}$–that is, the program counter and the backtrace of calling functions must be identical. *test* returns ✓ if the program exits without failure and ? in all other cases[Zel02].

partitioning of these sets also depends on having knowledge about the structure of the input. In general, the underlying assumption of the method is that the input is structured. Therefore, it is hard to see how the technique would be applied to arbitrary programs with generally unstructured inputs such as sets of numbers.

Delta Debugging in [Zel02] is used to isolate in every state of the program the relevant variables and values to the failure. Therefore, in this work, cause is defined in the *program state* and isolated by systematically minimizing the difference δ between a program state of the failing run and the corresponding state of the passing run. The underlying assumption in [Zel02] is that the given failing and passing traces are similar. For the running example of [Zel02], these traces were generated using similar inputs. However, the assumption that similar inputs produce similar traces is not necessarily true in general. Instrumentation points at which states from two traces are compared need to be given by the programmer. Moreover, not every two states from the failing and the passing traces can be compared. Only those states with identical program counter and backtrace of their corresponding locations are *comparable*. Comparison of states involves generating *memory graphs* containing typically several thousands of variables. To precisely determine which variables are added or deleted in a failing state (compared to a passing state), the largest common subgraph of the corresponding memory graphs needs to be computed, which is an NP-complete problem in general. Finally, although this method facilitates debugging, it does not explicitly reveal the faulty statements because it only detects the cause in individual states. To find the faulty statement, the cause-effect chain needs to be isolated by the programmer.

In [CZ02], for a multithreaded program failure-inducing thread schedules are isolated as the cause of failure. In this work, Delta Debugging is used for systematically narrowing down the difference between the thread schedules of a passing and a failing run until a minimal difference achieved. A thread schedule T is represented as an ordered list of n clock times: $T = \langle t_1, t_2, \ldots, t_n \rangle$ in which $t_i < t_{i+1}$ for all $1 \leq i \leq n$ and each t_i corresponds to a thread switch between the threads. To compare the thread schedules of a failing run with a passing run, the assumption is that both schedules have the same length. In case they do not have the same length the shorter schedule needs to be padded with "dummy" thread switches which would occur after the execution of the program in question ended. The difference or δ between the two schedules is defined as a set of *thread switch changes*, δ_i, each corresponding to the difference between the i^{th} thread switch of passing and failing thread schedules: $|t_{\checkmark i} - t_{\boldsymbol{\times} i}|$. δ_is are further decomposed into atomic changes $\delta_{i,1}, \delta_{i,2}, \ldots$ each narrowing the difference between $t_{\checkmark i}$ and $t_{\boldsymbol{\times} i}$ by one clock unit. Delta Debugging then isolates a minimal subset of $\delta_{i,j}$ which are relevant for the failure.

Considering the way $\delta_{i,j}$s are defined, the method seems to be applicable only if failing and passing traces are over the same set of events. In other words, failing and passing traces need to take the same execution path and differ only in scheduling. This requirement significantly limits the application of the method since generation of these traces involves several tests. Moreover, thread switches t_is are only represented by clock times and

they do not show the corresponding threads. Therefore, it seems that the assumption of the method is that corresponding thread switches of failing and passing traces are between the same threads which also restricts the choice of the input failing and passing traces. The number of elementary changes $\delta_{i,j}$ can quickly become very large because it is quadratic in the length of the schedules. Therefore, a significant number of tests are required for isolating the failure inducing difference.

Error Explanation with Distance Metrics

Based on Lewis' counterfactual causality reasoning, Groce et al. [GCKS06] developed a tool called EXPLAIN, which extends the CBMC[6] model checker [KCL04], for assisting users in understanding and isolating errors in ANSI C programs. Given an ANSI C program, the bounded model checker CBMC generates a set of constraints that encodes all executions of the program up to a certain loop unwinding depth. The loop unwinding depth determines the maximum number of times each loop may be executed. The representation of the constraints is based on static single assignment (SSA) form [CFR+91] and loop unwinding. By conjoining the set of constraints that encodes the executions of a program P up to a certain depth with the negation of the specification of P, CBMC produces a Boolean satisfiability formula, S. Using a SAT solver, CBMC then finds any satisfying assignment of S which represents a counterexample, a finite execution of the program that violates the specification. Given a counterexample, by solving an optimization problem EXPLAIN finds the most similar passing execution as measured by a *distance metric* on executions of P. The distance between two executions a and b, $d(a, b)$, is defined as the number of variables to which a and b assign different values. This is due to the fact that all executions of P are encoded as a series of assignments in the SSA form based representation. Moreover, all these executions (for a fixed loop unwinding depth) are represented as assignments to the same variables. The differences (Δs) between the most similar passing execution and the counterexample, after being refined by a slicing step, is given to the programmer as the cause of failure or the bug explanation.

Since sequential programs are deterministic, the only change EXPLAIN can make in searching for a passing execution is the input values. Consequently, Δs can contain assignments which are only due to different input values and are not relevant to the bug. The slicing step of the method is, in fact, for filtering such assignments from the bug explanations. Moreover, Δs does not necessarily reveal the faulty parts of the code. In [WYIG06], two code fragments are given for which EXPLAIN fails to isolate the faulty statements. One of these examples is shown in Listing 2.4 which is supposed to find the maximum of three inputs.

[6] http://cprover.org/cbmc/

```
1  find_max (x1, x2, x3)
2  {
3    max = x1;
4    ...
5    if (max <= x2)
6      max = x2;
7    ...
8    if (max >= x3)
9      max = x3;
10   ...
11   assert (max >= x1);
12   assert (max >= x2);
13   assert (max >= x3);
14 }
```

Listing 2.4: Computing the maximum of three inputs

The input (0,1,0) results in an execution that violates the assertion at Line 12. The buggy condition of Line 8 (it should have been max <= x3) causes the assertion failure. According to the distance metric used in Explain [GCKS06], the most similar passing execution to the counterexample can be generated by the input (0,0,0). Table 2.1 compares the variable assignments at different execution steps of the counterexample and the passing execution. Each row in the table shows the name of the variables or conditions, their corresponding program locations (specified with @), their values, and the distance as is measured in [GCKS06]. Since at Line 6 there is a different assignment

Table 2.1: Counterexample and passing executions for find_max

Variables/Conditions	Variable/Condition Valuations in		Distance
	Counterexample	Passing execution	
x1 @ 1	0	0	
x2 @ 1	1	0	1
x3 @ 1	0	0	
max @ 3	0	0	
(max <= x2) @ 5	true	true	
max @ 6	1	0	1
(max >= x3) @ 8	true	true	
max @ 9	0	0	
(max >= x1) @ 11	true	true	
(max >= x2) @ 12	false	true	

to max, Explain would classify Line 6 as cause of the failure. However, Line 6 and the condition at Line 5 are both correct, and the bug is at Line 8.

Halpern and Pearl Causality Framework

The naive interpretation of the Lewis counterfactual test leads to a number of inadequate or even fallacious inferences of causes. The following story, presented by Hall in [Hal04], demonstrates some of the difficulties in this definition. Suppose that Suzy and Billy both pick up rocks and throw them at a bottle. Suzy's rock gets there first, shattering the bottle. Since both throws are perfectly accurate, Billy's would have shattered the bottle had it not been preempted by Suzy's throw. Thus, according to the counterfactual condition, Suzy's throw is not a cause for shattering the bottle (because if Suzy wouldn't have thrown her rock, the bottle would have been shattered by Billy's throw). Since the counterfactual dependence does not capture all the subtleties involved in causality, Halpern and Pearl proposed a framework for causality analysis which is an extension of the Lewis' counterfactual model. They refer to this framework as structural equation model (SEM) [HP05, Hal15]. In Halpern and Pearl causality framework, roughly speaking, A is considered to be the cause of B if B counterfactually depends on A under some contingency. For example, Suzy's throw is a cause of the bottle shattering because the bottle shattering counterfactually depends on Suzy's throw, under the contingency that Billy doesn't throw.

At the heart of the structural model approach is the idea that causality can be represented in terms of functional relations between the variables of some domain. In this framework, the causal relations between a set of variables is represented by a set of structural equations. These structural equations express information about the effects of potential interventions. A *causal model* M is defined, then, as a set of variables together with a set of functional relations which encode the causal relationship between the variables. A *causal formula* ϕ is a Boolean combination of formulas of the form $X = x$ in which X is a variable and x is a possible value of this variable. The *actual cause* ("cause" for short) of a formula ϕ in the causal model M is of the form $X_1 = x_1 \wedge ... \wedge X_k = x_k$ in which each X_i where $1 \leq i \leq k$ refers to a variable and the corresponding x_i is a possible value of that variable. A formula of the form $X_1 = x_1 \wedge ... \wedge X_k = x_k$ is defined as the cause of ϕ in the causal model M under certain conditions. These conditions are referred to as $AC1$, $AC2$ and $AC3$ in [HP05, Hal15]. $AC1$ just says that A cannot be a cause of B unless A and B are both true in the causal model M. $AC3$ ensures the minimality of the cause. The core definition of cause lies in $AC2$. It has two parts: first is, in fact, a counterfactual test and the second restricts the causes satisfying the counterfactual test because as we discussed counterfactual test can be permissive. For a more detailed treatment we refer the interested reader to [HP05, Hal15].

The work in [BBDC+09] relate the formal definition of causality of Halpern and Pearl to finding the cause of failure in counterexamples. They adapt Halpern and Pearl definition of causality to the analysis of a counterexample π with respect to a temporal logic formula φ (for definition of temporal logic see [BK08]). They analyze counterexamples produced from hardware model checkers. They view a trace (counterexample) as a set of pairs $\langle location, signal \rangle$, and look for the pairs that are the causes for the failure of φ according to the definition of cause in [HP05, Hal15]. They argue that the causality

analysis of a counterexample π and a temporal logic formula φ can be modeled as a *binary causal model* in which the variables are Boolean. Therefore, they present the significantly simpler version of causality for binary causal models in which the counterexample π and the formula φ are considered as a binary causal model. In the counterexample trace $\pi = s_0, s_1, \ldots$, each s_i is a state of the hardware model which is a valuation of Boolean signals v. The cause of the failure of φ in π is in the form of a pair $\langle s, v \rangle$ where s is a state and v a Boolean signal. Since computing causality for binary causal models is NP-complete [EL02], they show that computing the set of causes for the failure of a linear temporal logic (LTL) (see [BK08] for syntax and semantics of LTL) on a single trace is also NP-complete. To improve scalability, they propose a polynomial-time algorithm for approximating the set of causes. The experimental evaluation performed on real-world examples demonstrate that the causes computed by the approximate algorithm match the user's intuition on all the examples.

In [LFL13], the structural equation model (SEM) by Halpern and Pearl is adapted to compute causality for counterexamples generated by model checking of concurrent systems. Since in concurrent systems event interleavings can be the cause of the error, (race conditions, for instance) the order of occurrence of events needs to be considered as a potential causal factor for failure. Therefore, the authors of [LFL13] extend the structural equation model to account for event orderings. In their adapted framework from [HP05, Hal15], they consider only Boolean variables, hence a binary causal model. The Boolean variables in this model represent occurrence of events. In order to express and reason about the order of the occurrence of events, the authors of [LFL13] define an *event order logic*. The logic uses a set of event variables which are Boolean and a set of Boolean connectives $\wedge$, $\vee$ and $\neg$. To express the ordering of events the logic uses the ordered conjunction operator $\owedge$. The formula $A \owedge B$ is satisfied if and only if events A and B occur in a trace and A occurs before B. The formal semantics of this logic can be found in [KLL11]. Using the event order logic, cause in [LFL13] is defined in the form of an event order logic formula, therefore, in this setting the order of occurrence of events can also be extracted as cause. For computing causes, they extend depth-first search and breadth-first search algorithms used for state space exploration in explicit state model checking. The main limitation of this work is that for computing the combination of events which are causal for a property violation, the method needs to consider all possible finite bad and good execution traces. The existential and universal quantifiers in the conditions defined in their causality framework show that for checking these conditions they need to consider all the possible finite executions of the system. (Since they use explicit state space model checker Spin [Hol03], they analyze only finite state models.)

Model-based Diagnosis

We have seen so far debugging approaches based on Lewis' counterfactual reasoning and Halpern and Pearl's structural equation model for causality analysis. There exist also approaches based on *theory of diagnosis* [Rei87] for automatically computing the cause of a failure in a program. The theory of diagnosis is a general theory which addresses

the diagnostic reasoning for finding the faulty components in a system [Rei87]. This theory is applicable to a wide variety of practical settings such as digital and analogue circuits, medicine and database updates. Moreover, the theory leads to an algorithm for computing all diagnoses. In this theory, the system is described in a suitable logic. There exists a failing *observation* of the system which is also represented as a logical formula. The failing observation is a system behavior which conflicts with what the *system description* predicts should happen if all its components were behaving correctly. By combining the system description with failing observation and the assumption that all the components of the system are correct, we get an inconsistent formula. A *diagnosis* is, then, defined as a conjecture that a *minimal* set of components are faulty so that by allowing these components to be faulty in the inconsistent formula mentioned above, the formula becomes satisfiable.

The theory of diagnosis underlies the method proposed in [JM11] for performing fault localization in C programs. Given a program, a test case which results in a failure and the corresponding failing execution, the method (which has been implemented in a tool called BugAssist) outputs a minimal set of program statements such that fixing them makes the failing execution disappear. Using the bounded model checking and a bound obtained from the execution of the failing test, they encode the semantics of a bounded unrolling of the program as a Boolean formula. Every satisfying assignment of this formula corresponds to a feasible program execution up to a certain unrolling depth. By conjoining this formula with a correctness specification (an assertion, or a "golden output") and a failing test case, they construct an unsatisfiable formula. This unsatisfiable formula is then given to a *maximum satisfiability solver*. Maximum satisfiability (MAX-SAT) is the problem of determining the maximum number of clauses of a Boolean formula which can be satisfied simultaneously by any given valuations [LM09]. The complement of a *maximum satisfiable subset* (MSS) is a set of clauses whose removal makes the formula satisfiable and are referred to as a minimum correction set (MCS). Since the maximum satisfiable subset (MSS) is maximal, the complement of this set (MCS) is minimal [LS05]. Using a MAX-SAT solver, BugAssist computes a maximal set of clauses of the unsatisfiable formula which can be satisfied simultaneously (MSS), and outputs the complement of this set (MCS) as a candidate cause of failure or a diagnosis. Since there may be several of these minimal sets of clauses, the tool enumerates each of them as a candidate cause or a diagnosis.

The main issue with the method of BugAssist in [JM11] is performance. As the experimental results show when the programs are not large, the tool performs efficiently. However, for larger programs (around 370-730 LOC) the corresponding MAX-SAT solver could not process the generated formulas. Therefore, they had to use trace reduction techniques such as program slicing or delta debugging to isolate failure-inducing inputs. Even after reducing the trace size, for a case study with a relatively long trace it took BugAssist around 11h to find the exact location of the bug. Moreover, as we have explained, BugAssist may report multiple causes of failure or fault locations although in the experiments of [JM11] this number remained small. Therefore, the authors of [JM11]

proposed running the localization algorithm repeatedly with different failing executions and based on that subsequently ranking candidate fault locations.

2.2 Debugging Concurrent Programs

So far we focused mostly on different approaches proposed for debugging sequential codes. In this section, we discuss concurrency bug detectors which are generally based on exploiting the characteristics of common types of *concurrency bugs*.

2.2.1 Concurrency Bugs - Concurrency Bug Detectors

In the following, we introduce the common types of concurrency bugs that arise in practice according to a study by Lu et al. [LPSZ08]. The study focuses on four major and important open source applications: MySQL (database management system), Apache (web server), Mozilla (web browser), and OpenOffice (a free version of the MS Office suite). The 105 bugs studied in [LPSZ08] are divided into two broad categories of *deadlock*(30%) and *non-deadlock*(70%) bugs. These two categories of bugs have different properties. In this dissertation, bugs from both categories are addressed.

Deadlock

A deadlock occurs when threads mutually wait in a non-preemptive fashion for each other to release an acquired shared resource, for example locks. Therefore, they prevent executions from making any progress. A simple deadlock example is given in Figure 2.3. In this figure, Thread1 holds Lock1 and waits for Lock2 while Thread2 holds Lock2 and waits for Lock1 (held by Thread1). Therefore, both threads are waiting for each other without making any progress. As we can see, there is a *cycle* in Figure 2.3 which is indicative of a deadlock. Deadlocks similar to the one depicted in Figure 2.3 can easily

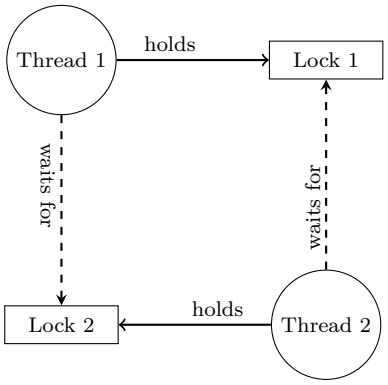


Figure 2.3: A simple deadlock example

be avoided by defining a locking discipline. For example, a locking discipline can be

```
/*- - - Thread 1 - - -*/
...
write(sharedVar);
...
```

```
/*- - - Thread 2 - - -*/
...
read(sharedVar);
...
```

Figure 2.4: Race condition: simultaneous memory accesses (at least one write)

choosing an order among all locks and making sure when threads grab more than one lock, they acquire them in the same order.

In addition to deadlock prevention mechanisms, there also exist techniques for detecting deadlocks. Static deadlock detection approaches are based on type systems [BLR02], dataflow analyses, or model checking [God97, MQ07, HP00]. Type-based approaches rely on significant annotations. Model checkers have scalability problems. Approaches based on dataflow analysis, although have been applied to large programs, are highly imprecise.

Data Races

Two memory access instructions from different threads *conflict* if both target the same memory location and at least one of them is a write operation. A data race occurs if two conflicting memory accesses happen concurrently, without synchronization (Figure 2.4). However, not all data races are erroneous. Benign data races include races that affect the program outcome in a manner acceptable to the programmer such as updates to logging/debugging variables [EMBO10].

Data race detection methods can be broadly classified into static and dynamic. Static race detectors [EA03a, NAW06] analyze the source code without executing the program. Static techniques are more prone to false positives because of relying on approximate information such as pointer aliasing. Dynamic race detection [SBN+97a, FF10, PS07] is based on instrumenting the program and monitoring the execution. Dynamic techniques produce less false positives than static techniques since they monitor an actual execution of the program. However, they may miss some races since they monitor a single execution.

Dynamic data race detectors are based on lockset analysis [SBN+97a], happens-before (hb) analysis [FF10, PS07], or a combination of the two [OC03]. A lockset is a set of locks which are held at a program location. In lockset analysis, locksets are computed for all locations in a program. A data race is detected when conflicting accesses from program locations with *disjoint* locksets occur. This type of analysis usually produces many false positives [EMBO10].

Happens-before analysis is based on *happens-before* relation (hb) which was first suggested by Lamport for message passing systems [Lam78]. The happens-before relation is defined for all memory access events (reads, writes) and synchronization events (release, acquire) that happen in an execution trace of a multithreaded program. If a and b are two events in an execution trace σ of a multithreaded program, a happens-before b in σ, denoted by $a \xrightarrow{hb}_{\sigma} b$, if a occurs before b and if: (1) a and b are two events issued by the same thread

in the program order, or (2) a and b are acquire and release events both operating on the same synchronization object.

If two operations in an execution trace are not related by the happens-before relation, they are considered *concurrent*. Therefore, happens-before imposes a partial order over the operations of an execution trace as we have seen before. The hb relation between the conflicting memory accesses can be computed based on synchronization points. In happens-before analysis [FF10, PS07], during a program execution the happens-before relation is computed and a data race is detected if conflicting accesses are not ordered by hb. The computation of hb is based on comparing the logical time stamps of accesses from different threads to the same shared variable. The happens-before analysis can be done precisely, but it incurs an overhead and moreover dynamic executions have limited coverage.

Freedom from data races, however, does not establish the correctness of a concurrent program since it does not guarantee the absence of atomicity violations, which constitute the predominant class of non-deadlock concurrency bugs [LPSZ08].

Atomicity Violations

Atomicity/serializability is a property of a sequence of program actions. A sequence of program actions is considered as *atomic* if other threads observe either the result from the execution of all of them or none of them, but not the partial result from the execution of some of them. Atomicity can be ensured by proper use of synchronization mechanism such as locking. *Atomicity violation* is a type of failure that occurs when an interference (context-switch) happens in a supposedly atomic block. If this block accesses a shared variable and its execution is interleaved by instructions in another thread that access the same shared variable, the interleaving may violate the atomicity of the block. Figure 2.5 shows two code fragments that non-atomically update the balance of a bank account (stored in the shared variable balance) by depositing or withdrawing given values. The example does not contain a data race, since balance is protected by the lock balance_lock. The interleaving that occurs in Thread 1 between read and write of balance at statements $S1$ and $S2$ causes a stale value to be written in balance at $S2$, "killing" the transaction of Thread 2 which updates balance at statement $S4$. Statements $S1$ and $S2$ need to be executed atomically which can be achieved by protecting them with the same lock similar to the atomic block in Thread 2 which contains $S3$ and $S4$. In fact, there exists no atomicity violation bug in Thread 2 since update of balance is done in an atomic block and no interleaving can occur between $S3$ and $S4$.

Atomicity violation bugs can involve more than one variable. The atomicity violation bug in the bank account example (Figure 2.5) involving a single variable is referred to as a single-variable atomicity violation in previous work [PVH12, LC09]. There exists also multi-variable atomicity violations in which occurrence of interleaving between the accesses to multiple variables in an intended atomic block can cause a failure. Figure 2.6

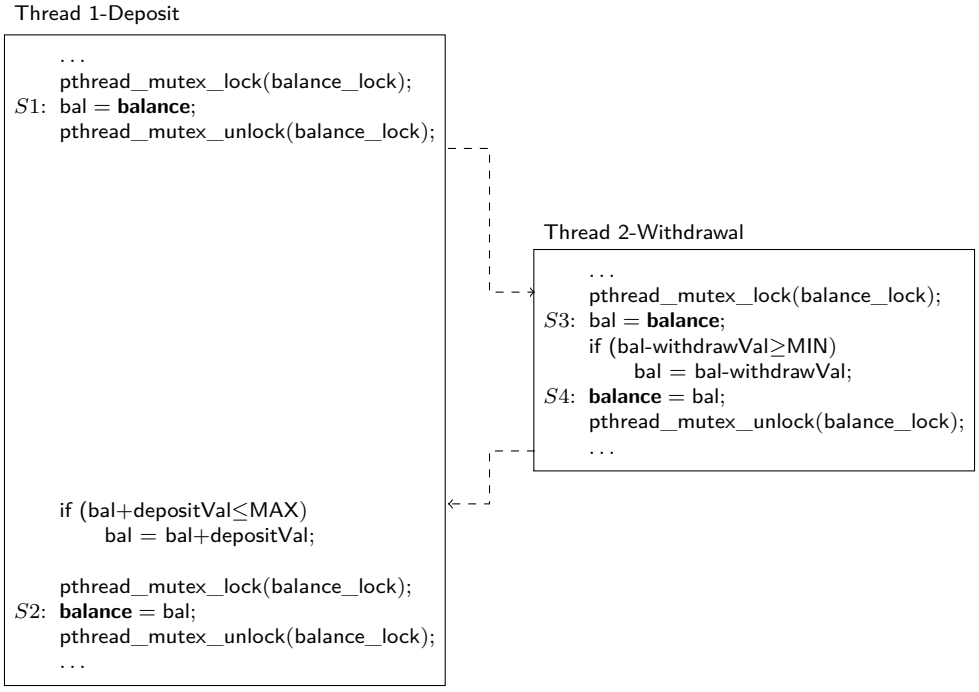


Figure 2.5: Atomicity violation in update of bank account balance (Single-variable Atomicity violation)

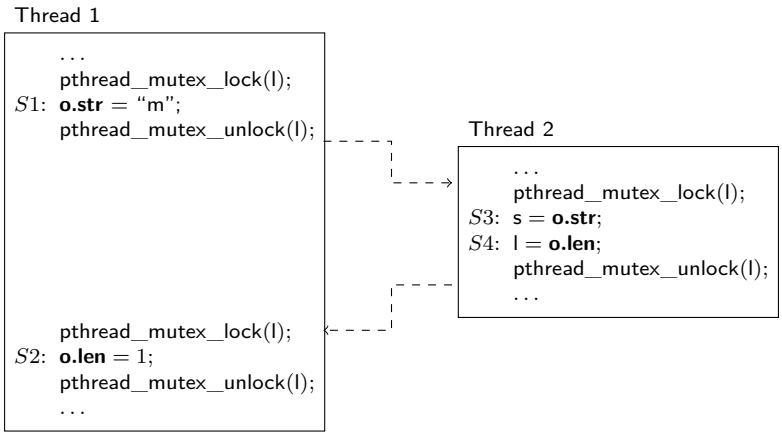


Figure 2.6: Multi-variable Atomicity violation

illustrates a multi-variable atomicity violation. In this figure, o is an object whose properties str and len are correlated, therefore they have to be updated atomically.

However, in Thread 1, updates of these two properties are not in the same critical region with the same lock. Occurrence of an interleaving after $S1$ causes values of these two properties read by Thread 2 to be inconsistent. While $S3$ in Thread 2 reads the new value of str (updated by $S1$ from Thread 1), $S4$ reads an old value of len since $S2$ has

not been executed at this point. The solution to this problem is protecting $S1$ and $S2$ with the same lock.

Various techniques both static and dynamic have been proposed for detecting atomicity violations. Artho et al. [AHB03] is among the first efforts to characterize atomicity violations for shared memory concurrent programs, referred to it as *high level data races*. In this work, for each thread, *views* which are sets of memory locations that are accessed in a single critical region (a block of code protected by a lock), are computed. Views of different threads are then compared. If one thread accesses data together in the same view and another thread accesses the same data separately in different views, a *view consistency violation* implying an atomicity violation is reported. The static analysis method of [FQ03] relies on a type and effect system for expressing and checking the atomicity of methods in a program. A limitation of this work is that it requires programmers to specify all synchronization points. Avio [LTQZ06] detects single-variable atomicity violations by learning memory access patterns from a sequence of passing training executions, and then monitoring whether these patterns are violated in subsequent runs. The patterns which Avio learns during training are pairs of memory accesses which are executed atomically. These patterns are called *access interleaving invariants* (AI). Avio can only detect atomicity violations involving pairs of accesses to the same memory location. The simple heuristic used in Avio leads to some false positives which are dealt with using post processing. Similar to Avio, AtomTracker [MOT10] infers atomic regions by observing passing executions during training phase. The inferred atomic regions are then used for identifying atomicity violations. Unlike Avio, AtomTracker consider atomic blocks larger than pairs of memory accesses. The accuracy of both AtomTracker and Avio depends on the training set which consists of only passing executions. Moreover, they do not take into account the *frequency* of observation of atomicity constraints in the training runs which can improve the learning process.

Svd [XBH05] is a tool that approximates atomic regions based on some heuristics, using control and data-dependencies, without relying on given atomic annotations. It then uses deterministic replay to detect atomicity violations. Atomizer [FF04] is a dynamic approach that uses modified lockset analysis to detect when the atomicity of atomic blocks is violated during an execution.

The main limitations of these techniques is their dependency on heuristics or atomicity annotations for obtaining atomicity constraints.

2.2.2 Concurrency Bug Explanation

Concurrency bug detectors we discussed so far address only one specific type of concurrency bug such as atomicity violation or data races. Moreover, many of these detectors have high false positive rates [FF04, SBN+97a], therefore accuracy is another challenge. There exists also large body of work on explaining or localizing concurrency bugs after they are triggered. The aim of these techniques is not to detect or trigger the concurrency bugs, but understanding the cause of the failure after a failing behavior is observed. In

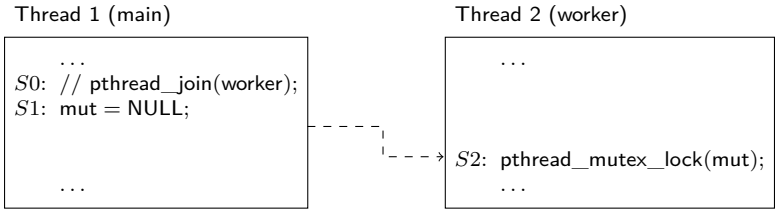


Figure 2.7: Order Violation (extracted from *PBZip2*)

addition, these techniques provide more general frameworks which also address *order violation*.

Order Violations

An *order violation* occurs when threads execute in an unexpected order which leads to a failure: two sequential accesses from two different threads to the same shared memory location where at least one of them is a write. Although pattern of memory access in order violation is similar to race condition, they are different as the cause in order violation is due to an unintended order of execution of threads. Figure 2.7 gives an example of an order violation. In this figure, since the main thread deinitializes the shared variable mut with a null value at $S1$ "before" the worker thread finishes, the context switch (depicted with dashed line) causes the program to crash with null-pointer exception at $S2$. This problem can be solved if the main thread waits for the worker thread to finish (statement $S0$) and then deinitializes mut with a null value.

Although data race detectors such as [SBN+97a, NAW06] can detect the order violation in Figure 2.7, recently more general frameworks for concurrency bug detection have been proposed that address non-deadlock bug types that we discussed so far.

Statistical methods are effective in localizing bugs in sequential programs (Section 2.1.2). Recently they have been applied for localizing concurrency bugs. Methods proposed in [LC09, LWC11, PVH10, PVH12] by statistically modeling program executions provide more general approaches to concurrency debugging. All these techniques collect shared memory accesses between threads during program execution and output a set of memory accesses ranked by suspiciousness.

Bugaboo [LC09] and Recon [LWC11] construct a form of *communication graph* called *context-aware communication graph* for detecting concurrency bugs. A communication graph encodes inter-thread data flow of a program execution in which nodes represent memory instructions and edges represent inter-thread communication via shared memory. An edge between two nodes representing memory instructions from two different threads, the *source* and the *sink*, shows that the sink instruction read or overwrote data written by the source instruction. Concurrency bugs may lead to edges that are only present or frequent in graph of failing executions, therefore graph differences can reveal problematic communications. Although this technique is often useful, but insufficient in general. This is because for some bugs (such as the one in Figure 2.6) the problematic edges are present

both in communication graphs of failing and passing executions. Authors of [LC09] demonstrated that basic communication graphs are insufficient for general concurrency bug detection. This inadequacy lead them to develop *context-aware communication graphs* which add access ordering information including the communication context to communication graphs. Communication context is a short (for example, 5 entries) history of type of shared memory operations performed in an execution and is maintained by each thread. Recon [LWC11] extends Bugaboo by reconstructing *temporal sequences* of communication events, and using machine learning to infer which sequences most likely explain a concurrency bug. The limitation of these two tools is that they rely on a bounded size context for bug detection and if the relevant ordering information is not encoded they may miss the bug.

Falcon [PVH10] and the follow-up work Unicorn [PVH12] monitor pairs of memory accesses in execution traces which are then combined into problematic patterns. These patterns are produced using a set of pattern templates for different bug categories including single- and multi-variable atomicity violations as well as order violations. Furthermore, Unicorn restricts these patterns to windows of some specific length, which results in a local view of the traces. The suspiciousness of a pattern is computed by comparing the number of times the pattern appears in a set of failing traces and in a set of passing traces.

The techniques discussed above start searching for the cause of the failure after the bug is triggered during testing of the program. For example, Unicorn [PVH12] executes each subject program 100 times and analyzes the resulting failing and passing traces. Bugaboo [LC09] collects 25 buggy runs and 25 non-buggy runs for further analysis.

CCI [JTLL10] proposes a statistical method for diagnosing production run failures caused by concurrency bugs. CCI monitors interleaving-related predicates at run time and it then uses statistical models to process run time information collected from many runs to identify the root causes of production run failures. To keep run time overhead low, CCI applies sampling strategies for collecting the values of predicates at run time.

The method proposed in [WWY+14] identifies buggy shared memory accesses as the cause of the failure in concurrent programs. Similar to Unicorn [PVH12], this method also uses an exhaustive list of pattern templates for order violations, single- and multi-variable atomicity violations. However, in contrast to Unicorn this method only requires a single failing run for comparison with a number of passing runs which makes the method more practical. This is due to the fact that capturing elusive failing runs is more difficult in practice.

ConcBugAssist [KKW15] is a logical constraint based symbolic method for diagnosis and repairing concurrency bugs. Similar to BugAssist [JM11] (Section 2.1.3), this tool encodes program behaviors up to a given bound using the bounded model checker CBMC. MAX-SAT solver is then used to compute a minimum subset of the inter-thread ordering constraints that are responsible for the failure which is the violation of an assertion. They show in the experiments that the set of inter-thread ordering constraints contained

in the diagnosis result of the tool represent on average a small fraction of the ordering constraints in the failing execution. Using the diagnosis result, ConcBugAssist also computes potential repairs which are modifications to the source code of the original program that are sufficient for eliminating the observed failure. Due to the dependency of ConcBugAssist on a bounded model checker, it faces scalability issues as the code size increases. In [KKW15], ConcBugAssist has been evaluated by applying it only to small programs whose lines of code are between 19 and 244.

In [GHR+15], SMT solver is used to compute succinct representation of concurrent error traces. Given an error trace, the method explores all the permutations of the events of the trace that correspond to feasible traces and lead to violation of the specification. The generalization of the discovered incorrect interleavings are represented as *HB-formulas* that are Boolean combinations of *happens-before* constraints between events. As an application, they use HB-formulas representing incorrect interleavings for bug summarization. The assumption of the authors in [GHR+15] is that HB-formulas which are constraints on scheduling are sufficient, for instance, for explaining concurrency bugs which are mostly due to bad ordering of instructions in a trace. While ConcBugAssist [KKW15] takes into account all the executions of the program up to a given bound, HB-formulas in [GHR+15] represent only reorderings of a fixed set of events.

2.2.3 Exposing and Reproducing Concurrency Bugs

The debugging work we discussed so far focuses on starting from a program failure, inferring the cause of the failure, and providing information for fixing the bug. Other prior work has focused on the related but orthogonal goal of *testing* or *model checking* concurrent programs to expose and reveal program failures. We include some of these techniques here because of the important link between testing or model checking (finding failures) and debugging (finding the cause of the failure and fixing them).

Exposing concurrency bugs require not only a bug triggering input but also a bug triggering interleaving. In theory, concurrency testing is the process of testing a concurrent program for correct behavior under all possible interleavings associated with every input. In practice, exploring the huge interleaving space of concurrent programs is infeasible. The common practice for exposing concurrency bugs has been *stress testing* which evaluates the behavior of a concurrent program under heavy load. Although stress testing indirectly increases the variety of thread schedules (interleavings), it is neither sufficient nor predictable since it may not cover bug triggering schedules and it cannot find the same error again. As we know concurrency bugs are considered as "Heisenbugs" which rarely surface and are hard to reproduce. Therefore, there is no guarantee that they can be revealed by stress testing.

Randomized scheduling is similar to stress testing, but increases the randomness of the OS scheduler by inserting random delays, context switches, or thread priority changes [BAEFU06]. It improves on stress testing by finding buggy schedules more effectively. Probabilistic concurrency testing (PCT), method presented in [BKMN10]

is a randomized algorithm which also quantifies the probability of missing concurrency bugs. PCT, in fact, provides a guaranteed probability of finding bugs in every run of the program.

Several different techniques have been proposed for systematically exploring the interleaving space [MQ07, FHRV13, YCGK07]. Chess [MQB07] is a successful concurrency testing tool based on iterative context-bounding, a search algorithm that systematically explores the executions of a multithreaded program (with a fixed input) in an order that prioritizes executions with fewer context switches [MQ07]. It distinguishes between *preempting* and *nonpreempting* context switches. A preempting context switch, or a *preemption*, occurs when the scheduler suspends the execution of the running thread at an arbitrary point, for example, at the expiration of a time slice. A nonpreempting context switch occurs when the running thread voluntarily yields its execution, either at termination or when it blocks on an unavailable resource such as a lock. Chess bounds the number of preemptions while leaving the number of nonpreempting context switches unbounded [MQ07]. Bounding the number of preemptions has several powerful and desirable consequences for systematic state space exploration of multithreaded programs such as scalability [MQ07]. Chess, in fact, replaces the OS scheduler with its own scheduler and runs the program several times with different scheduling choices.

Inspect [YCGK07] is a systematic concurrency testing tool which has been shown capable of detecting concurrency bugs such as data races, deadlocks, etc., in real-world PThreads C/C++ programs. It systematically explores all possible interleavings of a multithreaded C/C++ program under a specific testing scenario, employing dynamic partial order reduction (DPOR) (proposed in [FG05]). Partial order reduction (POR) techniques are used for combating the state space explosion problem by avoiding interleaving independent transitions during search in the state space of a concurrent system [CGP99].

Concolic (*conc*rete and symb*olic*) testing [GKS05] is a technique first introduced for testing sequential code. There, the set of possible inputs (the only parameter that can change for sequential programs) is systematically explored to provide standard code coverage guarantees, such as branch coverage, for the program. ConCrest [FHRV13] is a testing tool for concurrent programs which generalizes concolic testing to concurrent programs. The goal in ConCrest is achieving maximum code coverage for a concurrent program by systematically exploring both input and scheduling space. While Chess and Inspect for a particular input run the program to expose the failure inducing thread schedule, con2colic testing in ConCrest looks for the combination of an input and a schedule which causes a failure.

Tools such as Inspect and Chess are also referred to as runtime (or dynamic) model checkers since they check the correctness of the program by running the actual code. Verisoft [God97] and Java PathFinder [VM05] are other execution-based or runtime model checkers for concurrency bug detection in C/C++ and Java multithreaded programs, respectively.

Static model checkers have also been widely used for exposing bugs in concurrent systems.

In this type of model checkers, first a model of a concurrent system is extracted. The model checker then systematically checks whether the formal model of the system satisfies a formalized property [BK08]. SPIN [Hol03] is an explicit state space model checker which exhaustively explores the state space of a model in order to locate all property violating states. Therefore, it can be used for model checking finite state models such as communication protocols.

The testing and model checking tools only reveal full failing executions, and do not provide root cause information. Therefore, using these approaches does not free the programmer from understanding the cause of the failure and fixing it manually. Similarly, techniques proposed for recording and deterministic replay of executions in concurrent programs [LMC87, MCT08] allow only to reproduce a failing execution for further analysis, without providing useful information for debugging. For example, the Delta Debugging method of [CZ02] which isolates failure-inducing thread schedules as the cause of failure in multithreaded programs uses a tool called DEJAVU [ACN+01] for deterministic replay of executions of multithreaded java programs. During the execution of a java program, DEJAVU records the input and the thread schedule which will be used later for restoring the execution of the program deterministically.

CHAPTER 3

Counterexample Explanation by Anomaly Detection

As we have seen previously, the interleaving semantics commonly used to interpret the computation of concurrent systems imposes a total order on the execution of concurrent actions in a system. Concurrency is then interpreted as nondeterministic choices between different interleavings [BK08]. System designers are used to thinking sequentially when designing a system. In concurrent systems, it is therefore highly probable that they do not foresee some interleavings that their system encompasses. As a consequence, it is a widely held view that one of the main sources of failure in concurrent systems is unforeseen interleavings resulting in undesired behavior or unexpected results [BK08].

Model checkers are particularly well-suited for detecting concurrency bugs due to the exhaustive exploration of all possible interleavings of the concurrent actions that they perform. They can therefore reveal bugs which are impossible or difficult to find by testing methods. Model checking which is an established technique for the automated analysis of hardware and software systems, systematically checks whether a formal model M of the system satisfies a formalized property P [BK08]. If M contains a fault so that M does not satisfy P, as a symptom of the fault in the model, the model checker generates a counterexample to the satisfaction of P. Given that counterexamples are only symptoms of faults in the model, a significant amount of manual analysis is required in order to locate a fault that constitutes a root cause for the presence of the counterexample in the model. System designers need to inspect lengthy counterexamples of sometimes up to thousands of events in order to understand the cause of the violation of P by M. Since this manual inspection is time consuming and error prone, an automatic method for explaining counterexamples that assist system designers in localizing faults in their models is highly desirable.

In this chapter, we present an automated method for explaining counterexamples indi-

cating the violation of a desired property in concurrent systems. Our method benefits from the analysis of a large number of counterexamples that can be generated by a model checking tool such as Spin [Hol03]. When Spin explores exhaustively the state space of a model in order to locate all property violating states, it can generate a set of counterexamples. We refer to the set of counterexamples that show how the model violates a property, as the *bad dataset*. With the aid of Spin, it is also possible to produce a set of execution traces that do not violate the property. We refer to this set of non-violating traces as the *good dataset*.

Our method for explaining counterexamples is an anomaly detection technique based on examining the differences in the traces of the good and bad datasets. The anomalies produced by our method, which are given in the form of *sequences* of actions, can reveal to the system designer unforeseen interleavings that lead the system to a failure.

3.1 Counterexamples, Anomalies

In this section, we define basic notions such as counterexamples, system models, traces, and properties. We, then, introduce the notion of anomalies for counterexample explanation. We recap the terminology of sequential pattern mining and explain how we apply this technique to extract anomalies from sets of traces.

3.1.1 System Models and Counterexamples

Our goal is to explain the violation of a safety property in a concurrent system. Such a violation represents that there exist undesired or unsafe states which are reachable by system executions. We use the explicit state model checker Spin [Hol03] in order to compute system executions that lead from an initial state of the system into a property violating state, often referred to as counterexamples. In the following, we formalize the notion of the model of a concurrent system which is expressed in *Promela* the modeling language of Spin [Hol03].

A model of a concurrent system expressed in Promela consists of a finite number of processes, each referred to as a *proctype*[1], and a set of shared variables $\mathbb{V}$. Every proctype is represented by a control flow automaton (CFA) $\langle L_p, T_p, \lambda_p, L_{0_p} \rangle$, where L_p and T_p are sets of nodes and edges, respectively, L_{0_p} the initial node, and λ_p is a labeling function linking edges to *basic statements*. The set of basic statements in Promela which we denote by *Act* comprises six elements: assignments, assertions, print statements, send or receive statements, and conditionals (or expressions) [Hol03]. Send and receive statements are communication instructions in the form of "$c\,!\,val$" and "$c\,?\,var$", respectively, where c is a *channel* (a buffer with a specific size), *val* a value (an expression), and *var* a variable(or a constant). In CFA, the set of nodes L_p corresponds to the points of control within the proctype, and the set of edges T_p defines the flow of control. The edge labels defined

[1] *proctype* is the keyword used in Promela for defining a process.

by λ_p are the basic statements which specify either the executability or the effect of execution of an edge.

A state s is a mapping from the variables of system (including $\mathbb{V}$ and local variables of proctypes) to values. If an edge is labeled with an expression ϕ, the valuation of ϕ in the system state determines the executability of that edge. Otherwise, the effect of executing an edge on the system state is according to the semantics of the basic statement that the edge is labeled with. (We refer the interested reader to [Hol03] for the semantics of basic statements.) Figure 3.1 shows an example of a CFA. Note that for nodes with more than one outgoing edge, more than one edge can be executable simultaneously (for modeling nondeterminism).

The semantics engine in SPIN executes a Promela model in a step by step manner, selecting one executable statement in each step. If more than one statement is executable, one of them is selected, for instance, randomly for execution. The semantics engine continues executing statements until no executable statements remain [Hol03]. The behavior of a model which is the sequence of states observed during the execution is formalized using the notion of *executions*. An execution results from the resolution of the possible nondeterminism in the model. In concurrent systems, an execution is mainly the result of the nondeterministic choice between the concurrent actions, referred to as interleaving. The semantics engine in SPIN implements an interleaving semantics for resolving the nondeterminism.

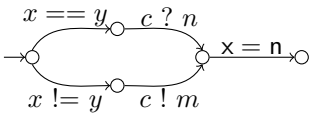


Figure 3.1: Sample control flow automaton (CFA) for a model in Promela

Definition 1 (Execution)**.** *An execution ρ corresponding to an interleaving of basic statements from proctypes of a system model, is an alternating sequence of states s and basic statements $\alpha \in Act$: $\rho = s_0, \alpha_1, s_1, \alpha_2, \ldots, \alpha_n, s_n, \ldots$ such that for all $0 \leq i$ the execution of α_{i+1} in state s_i leads to state s_{i+1} which is denoted by: $s_i \overset{\alpha_{i+1}}{\rightarrow} s_{i+1}$.*

Counterexamples. We use linear temporal logic (LTL) [BK08] to specify properties and we use $\sigma \not\models \varphi$ to express that a system execution σ violates an LTL property φ. Executions which violate a property are referred to as *counterexamples*. In our setting, we ignore the states visited during an execution and focus only on the statements $\alpha \in Act$ which are executed. Therefore, we define a trace as follows:

Definition 2 (Trace)**.** *A trace $\sigma = \langle e_1, e_2, ..., e_n \rangle$ is a finite sequence of events that corresponds to an interleaving of basic statements from proctypes of a system model. (Each e_i corresponds to the execution of a basic statement $\alpha \in Act$ in the system model.) A trace σ can correspond to several executions, however it is considered as* feasible *if it has at least one corresponding execution.*

In our setting, we formally define the events of a trace as follows:

Definition 3 (Event)**.** *An* event *is a tuple* $\langle \mathsf{id}, \mathsf{pid}, \mathsf{loc}, \mathsf{type} \rangle$*, where* id *is an identifier,* pid *and* loc *are the proctype identifier and the program location of the corresponding action,* $\mathsf{type} \in$ *Act is the corresponding action from the set of basic statements of Promela.*

Two events have the same identifier id if they are issued by the same proctype and agree on the program location, and the type. However, each event in the trace is unique. Therefore, two events with the same id are distinguished by their index in the sequence of a trace. As Definitions 2 and 3 show, we ignore the values of variables.

When we use the terms *counterexample* or *trace*, we refer to Definition 2. Note that executions of a system model can possibly be infinite (Definition 1), however we only consider finite executions in our analysis, hence a trace is defined as a finite sequence of events in Definition 2. This is due to the fact that counterexamples showing the violation of safety properties are finite execution fragments.

Since our method is based on anomaly detection, we need a set of counterexamples as well as a set of correct or non-violating traces for comparison. We refer to a set of counterexamples violating a given property φ as a *bad dataset*, denoted by Σ_B: $\Sigma_B = \{\sigma \mid \sigma \not\models \varphi\}$. The *good dataset* comprises the correct or non-violating traces that satisfy φ. Such traces can be generated by producing counterexamples to $\neg\varphi$. This is justified by the following lemma:

Lemma 3.1.1. *For an execution* ρ*, if* ρ *satisfies* φ*, denoted as* $\rho \models \varphi$*, then it holds that* $\rho \models \varphi \Leftrightarrow \rho \not\models \neg\varphi$ *[BK08].*

If φ is a safety property, the negation of this property yields a liveness property. The counterexamples violating a liveness property are infinite lasso shaped traces.

Definition 4. *Let* $\hat{\phi}$ *and* ϕ' *denote finite trace fragments. We call* $\phi = \hat{\phi}.(\phi')^{\omega}$ *an infinite lasso shaped trace where* $\hat{\phi}$ *is the finite prefix of* ϕ *and* ω *denotes the infinite repetition of* ϕ'*.*

For the purpose of our analysis we produce a finite good trace from an infinite good trace ϕ by concatenating $\hat{\phi}$ with one occurrence of ϕ': $\phi_{fin} = \hat{\phi}.\phi'$. We refer to a set of such traces as a *good dataset*, denoted by Σ_G: $\Sigma_G = \{\phi_{fin} \mid \phi \models \varphi\}$

3.1.2 Explanatory Sequences

As we have discussed in Section 1.2.2 of Chapter 1, in order to understand the cause of the failure in a counterexample (failing trace), we need to isolate inside the counterexample, the cause-effect chain which reveals a combination of events relevant to a property violation. Although in sequential programs a single event can be causal for a failure, in concurrent systems the specific *order* between the occurrence of at least two events from two different threads triggers the error. In a concurrent system, therefore, the order of

events inside a counterexample can also be causal for the occurrence of a failure and can hence point to a bug.

In this work, we explain concurrent counterexamples by identifying ordered sequences of events inside the counterexamples which are relevant to the failure. We refer to such sequences revealing specific orders between concurrent events inside a counterexample which are presumed to be causal for the property violation as *explanatory sequences*. We use *anomaly* detection for isolating explanatory sequences (Section 3.1.3).

A Motivating Example

Using an example case study we illustrate how a deadlock can occur due to the specific order of execution of a set of actions in the model of a concurrent system. The model we use in this example is taken from the BEnchmarks for Explicit Model checkers (BEEM) [Pel06]. It is a Real-time Ethernet protocol named Rether. This protocol is a contention-free token bus protocol for the data-link layer of the ISO protocol stack. Its purpose is to provide guaranteed bandwidth, deterministic and periodic network access to multimedia applications over commodity Ethernet hardware. The Promela code of this model consists of three proctypes:

1. The Node proctype, which corresponds to a node in the protocol. It communicates with the Token and Bandwidth proctypes in order to access the bandwidth slots. In our example, only two instances of Node proctype, which are named Node_0 and Node_1, are created at run time.

2. The Bandwidth proctype, which manages the access of the nodes to the real-time transmission. It allocates and frees the real-time transmission slots upon receiving reserve and release messages from the nodes.

3. The Token proctype, which guarantees deterministic and periodic access to the bandwidth by handing in a token to the nodes in turn.

In order to make the original model taken from [Pel06] smaller and simpler, we have reduced the values of its parameters as follows:

N = 2	Number of the nodes
Slots = 3	Number of slots (a bandwidth)
Real-time slots = 1	Maximum number of slots for real-time transmission (should be smaller than Slots)

In Figure 3.2, the last 33 events of a counterexample with 72 events which shows how the Rether model goes to a deadlock state are given. Each column in Figure 3.2 represents the events belonging to one proctype whose name appears as the title of the column. These events correspond to the execution of Promela statements of the 4 proctypes of the model.

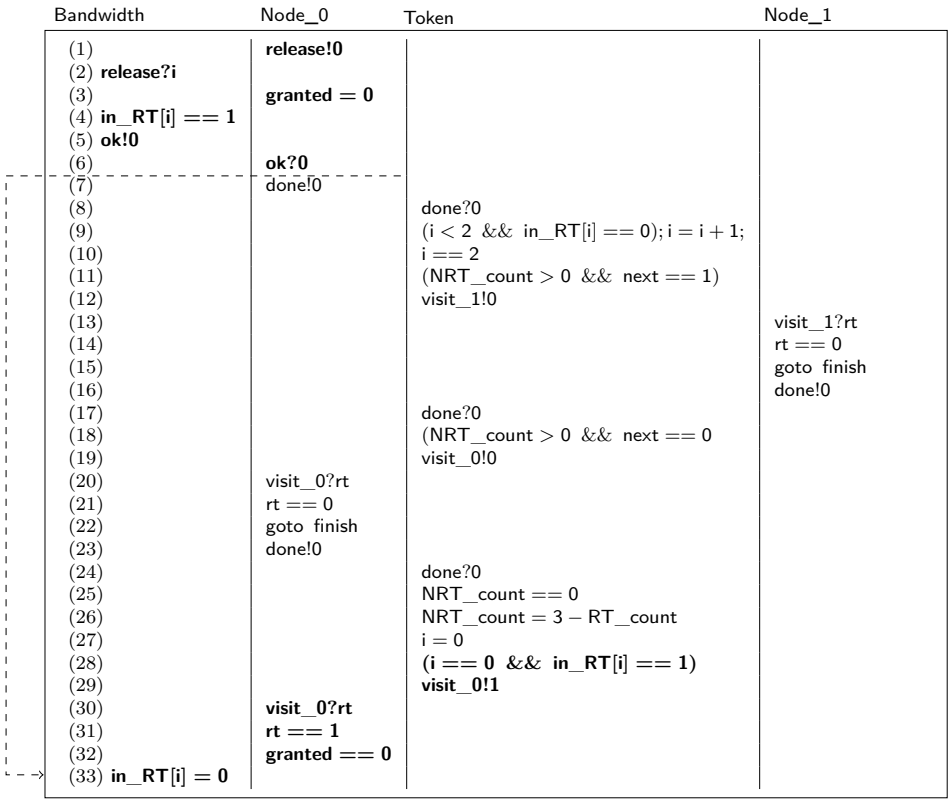


Figure 3.2: The last 33 events of a counterexample for the Rether protocol

By manual inspection and following the data- and control-dependencies we can identify a sequence of 12 events in the counterexample that explain the occurrence of the deadlock. These 12 events are highlighted by displaying them in bold font in Figure 3.2 and correspond to events 1-6 and 28-33. In order to understand how this sequence leads the system into a deadlock state we need to inspect the parts of the Promela code of the model which include the statements corresponding to the 12 events identified above. These parts are given in Figure 3.3 in which the numbers inside parenthesis refer to the corresponding event from the spotted 12 events from Figure 3.2. After Node_0 requests from Bandwidth the release of the corresponding real-time slot, in_RT[0], by sending release ! 0 (event 1), Bandwidth fails to do two correlated actions in an atomic step. These two actions are sending an acknowledgment back (ok!0, event 5), and freeing the corresponding real-time slot (in_RT[0] = 0, event 33). Due to scheduling there is a gap of 26 events between these two correlated events which results in a deadlock state. In the following, we explain the problem in more detail.

After the occurrence of events 5 (Line 11 of Bandwidth) and 6 (Line 29 of Node_0) in Figure 3.2, Lines 12 and 25 from Bandwidth and Node_0, respectively, become enabled simultaneously. As the trace in Figure 3.2 shows, among the two enabled statements Line 25 of Node_0, which corresponds to event 7 in this figure is chosen for execution.

Following the execution of event 31 in Figure 3.2 corresponding to Line 10 of Node_0, control is transferred to Line 14 of this proctype which is an if statement. Lines 15 and 16 of this if statement are both enabled simultaneously since Line 16 is a goto statement, and the guard of Line 15, granted == 0, is true. This is because the value of granted is set to zero at event 3 in Figure 3.2 and remains unchanged up to event 31. As Figure 3.2 shows, Line 15 of Node_0, which corresponds to event 32 in this figure, is executed. After execution of this line, Node_0 blocks and cannot take any action as the Promela code of this proctype in Figure 3.3 shows. At this point, Bandwidth also blocks waiting for a node proctype, Token blocks waiting for Node_0, and Node_1 blocks waiting for Token. This circular waiting results in a deadlock state.

One interesting characteristic of the identified sequence in Figure 3.2 is that the 12 events belonging to it do not occur adjacently inside the counterexample. While the first and the last six events occur next to each other, between these two groups of events there is a gap of 21 events. This is due to the nondeterministic scheduling of concurrent events due to the interleaving semantics implemented in Spin. As we have seen above, although Line 12 of the Bandwidth proctype was enabled after event 6, due to the nondeterministic execution of concurrent actions its execution is deferred to step 33. Dashed line in Figure 3.2 illustrate the gap between the position in the trace in which the statement Bandwidth.in_RT[i] = 0 becomes enabled, and the position in which it is actually executed.

The identified sequence in Figure 3.2 explaining the deadlock is an example of an *explanatory sequence* which reveals an unforeseen interleaving. The presumed intention of the model designer is that statements 11 and 12 of Bandwidth to be executed in an atomic step, which means they could not be interleaved with the actions of other proctypes. However, the proctype was implemented in a faulty way, so that its concurrent execution with other proctypes allowed the two mentioned statements to be executed nonatomically, and hence lead to a deadlock.

3.1.3 Mining Explanatory Sequences

As we have seen above, in an interleaved trace of concurrent events, the events belonging to a sequence which reveals an unforeseen interleaving do not necessarily occur next to each other. To the contrary, they can occur at an arbitrary, unbounded distance from each other, due to scheduling. The explanatory sequences are therefore, in general, *subsequences* of counterexamples. We define, formally, a subsequence relationship amongst sequences as follows:

Definition 5. *A sequence* $\pi = \langle p_0, p_1, p_2, ..., p_m \rangle$ *is a* subsequence *of another sequence* $\sigma = \langle e_0, e_1, e_2, ..., e_n \rangle$*, which is denoted by* $\pi \sqsubseteq \sigma$*, if there exist integers* $0 \leq i_0 < i_1 < i_2 < i_3 ... < i_m \leq n$ *where* $p_0 = e_{i_0}, p_1 = e_{i_1}, ..., p_m = e_{i_m}$*. We also call* σ *a* super-sequence *of* π*:* $\sigma \sqsupseteq \pi$*.*

Notice that a subsequence is not necessarily contiguous in the super-sequence. To capture the concept of a contiguous subsequence we introduce the notion of a *substring*:

Definition 6. $\psi = \langle q_0, q_1, q_2, ..., q_m \rangle$ *is a* substring *of* $\sigma = \langle e_0, e_1, e_2, ..., e_n \rangle$*, if and only if there exist consecutive integers from* $0 \leq i_0$ *to* $(i_0 + m) \leq n$ *such that* $q_0 = e_{i_0}, q_1 = e_{i_0+1}, ..., q_m = e_{i_0+m}$.

In our motivating example (Section 3.1.2), the explanatory sequence consisting of 12 events is a subsequence of the counterexample. However, it has two portions, events 1-6 and events 28-33, each is a substring of the original counterexample. Since explanatory sequences appear as subsequences of couterexamples, in order to isolate them via anomaly detection we need to extract subsequences which are only common or *frequent* in the bad dataset. Let FS_{Σ_B} and FS_{Σ_G} denote the set of *frequent* subsequences in the bad Σ_B and the good Σ_G datasets, respectively. For counterexample explanation we extract *anomalies* as defined in Equation 3.1. According to this equation, anomalies are subsequences that are frequent in the bad dataset Σ_B but not in the good dataset Σ_G.

$$anomalies \;=\; \mathrm{FS}_{\Sigma_B} \setminus \mathrm{FS}_{\Sigma_G} \tag{3.1}$$

To detect anomalies we use a standard pattern mining algorithm called *sequential pattern mining* which extracts the frequent subsequences from a dataset of sequences without limitations on the relative distance of events belonging to the subsequences [AS95] [DP07]. This data mining technique has diverse applications in areas such as the analysis of customer purchase behavior, the mining of web access patterns or motifs in DNA sequences. Frequent subsequence mining is an active area of research and a number of algorithms for mining frequent subsequences have been developed which have been proven to be efficient in practice with respect to various test datasets [YHA03, WH04, PHMA$^+$01].

Sequential Pattern Mining

We now give a brief overview of terminology used in sequential pattern mining, for a more detailed treatment we refer the interested reader to the cited literature and in particular to [DP07].

A sequence dataset S, $\{s_1, s_2, ..., s_n\}$, is a set of sequences. The $\mathsf{support}$ of a sequence p in S is the number of the sequences in S that p is a subsequence of: $\mathsf{support}_S(p) = |\{s \mid s \in S \wedge p \sqsubseteq s\}|$. Given a minimum support threshold, $\mathsf{min\_supp}$, the sequence p is considered a *sequential pattern* or a *frequent subsequence* if its $\mathsf{support}$ is no less than $\mathsf{min\_supp}$: $\mathsf{support}_S(p) \geq \mathsf{min\_supp}$. We denote the set of all sequential patterns mined from S with the given support threshold $\mathsf{min\_supp}$ by $\mathrm{FS}_{S,\mathsf{min\_supp}}$, i.e., $\mathrm{FS}_{S,\mathsf{min\_supp}} = \{p \mid \mathsf{support}_S(p) \geq \mathsf{min\_supp}\}$.

By contrasting the sequential patterns of the bad and the good datasets, we can extract patterns that are only frequent in the bad dataset. These patterns that are only frequent or common in the bad dataset, reveal anomalies, and hence can be indicative to the cause

of failure in concurrent system executions. However, as we will argue in the following, this at first sight promising tool fails due to the inherent complexity of the problem.

3.1.4 Challenges in Mining Explanatory Sequences as Sequential Patterns

In general, it can be shown that the problem of mining sequential patterns from a dataset of sequences is NP-hard. The complete proof is given in [Yan04], [Yan06]. Here, we provide a proof sketch.

Proof Sketch of NP-hardness

In order to show that the *sequential pattern mining* problem is NP-hard, it is sufficient to prove that the *frequent itemsets mining* problem [Goe03], which is the problem of mining frequent *itemsets* from a dataset of *transactions*, is NP-hard. This is because the latter problem can be reduced to the former one.

NP-hardness of Frequent Itemset Mining. Let $\mathcal{I}$ be a set of items. The set $X = \{i_1, i_2, \ldots, i_k\}$ where $i_l \in \mathcal{I}, 1 \leq l \leq k$ is called an *itemset* over $\mathcal{I}$. A *transaction* over $\mathcal{I}$ is a couple $T = (tid, I)$ where tid is the transaction identifier and I is an itemset over $\mathcal{I}$. A *transaction dataset* $\mathcal{D}$ over $\mathcal{I}$ is a set of transactions over $\mathcal{I}$. The $\mathsf{support}$ of an itemset X in $\mathcal{D}$ is defined as: $\mathsf{support}_{\mathcal{D}}(X) = |\{tid \mid (tid, I) \in \mathcal{D},\ X \subseteq I\}|$. An itemset is called *frequent* if its support is no less than a given minimum support threshold.

The transaction dataset $\mathcal{D}$ can be represented as a *bipartite graph* $G = (U, V, E)$ in which U and V are the two distinct vertex sets of G, and E is the set of edges. In this representation, U corresponds to the set of items $\mathcal{I}$, and V corresponds to the set of transaction identifiers: $\{tid \mid (tid, I) \in \mathcal{D}\}$. The edge set $E = \{(u, v) | u \in U \text{ and } v \in V\}$ represents all the (i, tid) pairs where $i \in \mathcal{I}$ is an item and tid a transaction identifier.

The problem of enumerating all maximal frequent itemsets from a transaction dataset $\mathcal{D}$ corresponds to the task of enumerating all maximal *bipartite cliques* in a bipartite graph G. A bipartite clique is a complete bipartite subgraph of a bipartite graph. Determining the number of maximal bipartite cliques in a bipartite graph is a *#P-complete* problem [Val79]. #P-completeness is used to capture the notion of the hardest counting problems, just as the concept of NP-completeness characterizes the hardest decision problems.

The above complexity arguments are based on worst-case complexity considerations [Yan04]. A number of sequential pattern mining algorithms have been developed that have proven to be efficient in practice with respect to various test datasets [AS95], [WH04], [PHMA+01]. However, the datasets that these algorithms have been evaluated on are sparse (w.r.t sequence length and number of sequence elements), with an average sequence length of less than 100. The densest dataset that an efficient sequential pattern mining algorithm, BIDE, can mine with a high support threshold of 90% has an average sequence length of 258 [WH04].

Model	#Trace		Avg. Trace Len.		Max Trace Len.	
	Bad ds.	Good ds.	Bad ds.	Good ds.	Bad ds.	Good ds.
Brp	660	25671	5985	5580	10539	10501
Rether	1061	26249	73263	63201	134629	134629
lann	989	20838	5737	6369	12612	12617
gear	614	10174	1994	3837	4512	4547
POTS	4109	107029	3006	5386	7988	8482
train-gate	222	3798	334	675	907	942

Table 3.1: Dataset characteristics

The characteristics of the bad and the good datasets of a number of Promela modeling case studies of concurrent systems are given in Table 3.1. In this table, the first four case studies are taken from [Pel06]. (The names of the corresponding Promela files for these case studies are given in Table 3.2.) The POTS model was developed by us as a sample model with numerous deadlock problems. This model is a non-trivial example of a telephony switch which comprises four concurrently executing proctypes corresponding to two users and two phone handlers. Each user in this model talks to a phone handler for making calls. The phone handlers are communicating with each other in order to switch and route user calls. In Table 3.1, the column "#Trace" gives the number of the traces in the bad and the good datasets and the columns "Avg. Trace Len." and "Max Trace Len." represent average and maximum length of the traces in these datasets, respectively.

It can be inferred from Table 3.1 that the bad and the good datasets are highly dense with the average sequence length of more than 1000 (except for the "train-gate" case study). We conclude that mining sequential patterns from the dataset of counterexamples generated from typical concurrent system models is intractable due to lengthy sequences and dense datasets.

3.2 Mining Substrings

To address the complexity challenges we encountered in mining sequential patterns from the bad and the good datasets, we abandon the feature of arbitrary distance between the events of a subsequence that we consider to reveal anomalies pointing at the causes of failures. As an approximation, we extract sequences that consist of consecutive events. Therefore, these sequences are substrings (Definition 6) of the execution traces contained in the good and bad datasets. Even though, as we have seen in the example of Section 3.1.2, a sequence that explains how a deadlock occurs is not necessarily the substring of a counterexample, it may contain portions which actually occur as substrings of a counterexample. In the example of Section 3.1.2, the sequences consisting of events $\langle 1, 2, 3, 4, 5, 6\rangle$ and $\langle 28, 29, 30, 31, 32, 33\rangle$, which are portions of explanatory sequence for the occurrence of a deadlock, are substrings of the counterexample. As we will explain in

this Section, by focusing on substrings which only occur in a set counterexamples, we will be able to provide anomalies which give hints at why a property is violated.

Therefore, we replace the definition of anomalies in (3.1) with a new definition in (3.2). According to this definition, *anomalies* are the substrings of length ℓ in the bad dataset which do not belong to the substrings of length ℓ in good dataset.

$$anomalies \;=\; S_{\Sigma_B,\ell} \setminus S_{\Sigma_G,\ell} \tag{3.2}$$

where $S_{\Sigma_B,\ell}$ and $S_{\Sigma_G,\ell}$ refer to the substrings of length ℓ in the bad Σ_B and the good Σ_G datasets, respectively. The length of the substrings, ℓ, which is the parameter of the method, can take various values. Since substrings of length ℓ can be extracted from a sequence of length n in $O(n)$ time, we avoid scalability problems. As we will see when presenting the experimental evaluation, the small value of $\ell = 2$ is adequate for explaining counterexamples using a fairly large set of case studies. To further justify this point, consider how a relatively short substring of length two can be indicative for the cause of a deadlock occurrence. For our running example in Section 3.1.2, the substring $\langle 31, 32\rangle$ occurs only in the set of counterexamples. Although $s = \langle 31, 32\rangle$ is only a small part of the spotted sequence which explains the occurrence of deadlock, $\langle 1, 2, 3, 4, 5, 6, 28, 29, 30, 31, 32, 33\rangle$, s can greatly facilitate identifying the other ten events of this explanatory sequence in the counterexample. In particular, the substring $s = \langle 31, 32\rangle$ shows that the variables Node_0.rt has the value 1 and Node_0.granted has the value 0, respectively at the same time. The statements which affect the values of these two variables, can be easily found in the counterexample. The value of the variable Node_0.granted becomes 0 at step 3 and remains unchanged until the end of the trace. The value 1 of the variable Node_0.rt is due to the value 1 of the global variable in_RT[0] while the value of this variable should be changed to 0 immediately after the variable Node_0.granted becomes 0.

Since due to scalability issues explanatory sequences revealing the cause-effect chains in counterexamples could not be extracted, we extract anomalies in the form of substrings of counterexamples, and show that they can greatly facilitate the search for explanatory sequences. The following sections describe in detail the steps of our method.

3.2.1 Generation of the Good and the Bad Datasets

For generating the good and the bad datasets, we use the explicit state Spin model checking tool [Hol03]. The default search algorithm that Spin uses for the exhaustive exploration of the state space is depth first search. When Spin locates the first violating state, it stops the search and reports the path from the initial state to the violating state as a counterexample. The presence of one counterexample is sufficient to show that the model does not comply with the specification.

There is also an option in Spin to not stop the search after locating the first violating state [Hol03]. With this option, "-c0 -e", Spin continues the search up to a given depth limit or until all states have been reached in order to locate all property violating states.

Our current strategy for generating the bad dataset is to use this option of Spin in order to explore exhaustively the state space of the model and to detect all the violating states and their corresponding counterexamples. Since the default depth limit in Spin is 10,000, we increase the depth limit until we can be certain that the state space has been exhaustively explored. Since DFS is used by Spin for exploring the state space, each violating state is visited once and so only one counterexample per violating state is generated.

Since the bad dataset contains the traces that violate some φ, the good dataset should include the traces that satisfy φ. Such traces can be generated by producing counterexamples to $\neg\varphi$ because a counterexample that shows the violation of the negation of a property actually satisfies that property according to Lemma 3.1.1.

The reachability property we consider in this work is deadlock-freedom, therefore we need to find a way to formalize the negation of that property in Spin. Notice that while the absence of deadlock is a safety property, its negation, which claims the presence of deadlocks, is a liveness property. As a consequence, the counterexamples to the presence of deadlocks are *lasso-shaped* infinite traces [Hol03](Section 3.1.1).

We specify the "presence of deadlock" property in Promela, the modeling language of the Spin model checker, by using a special state predicate named *timeout*. This predicate becomes true when the system blocks, i. e., when no statement in the model can be executed. We then specify the "presence of deadlock" property as *always eventually there will be a deadlock*, which can be expressed as requiring that always eventually the timeout predicate will become true. Spin tries to generate a counterexample for this property. The resultant counterexample will be a lasso-shaped infinite trace that never deadlocks. For the generation of the good dataset we also use the Spin option to not stop the search after generating the first counterexample for this property. Note that traces in the bad and good datasets are modeled according to Definition 2, and the events of the traces according to Definition 3.

3.2.2 Contrasting Sequence Sets

Substrings of length ℓ can be extracted from a trace by sliding a window of size ℓ over it. Figure 3.4 shows the nine possible substrings of length two that can be extracted from a trace of length 10 by sliding a window of size two over it. This set of substrings of length two, in fact, shows all the pair of events which occur next to each other in a trace. We refer to a set of all substrings of length ℓ of a trace as a *sequence set* of length ℓ of that trace.

We define sequence sets formally as follows (as proposed in [DLZ05] in another setting, see Section 2.1.2):

Definition 7 (Sequence Set)**.** *Let $\sigma = \langle e_1, e_2, \ldots, e_n \rangle$ denote a trace. The sequence set of length ℓ of σ, denoted by $ss_{\sigma,\ell}$, is the set of all substrings of length ℓ of σ: $ss_{\sigma,\ell} = \{\psi | \psi$ is a substring of σ and $|\psi| = \ell\}$*

As another example, consider $\sigma = \langle a, b, c, a, b, c, d, c \rangle$ and $\ell = 2$. The resulting sequence set of length ℓ of σ is: $ss_{\sigma,\ell} = \{\langle a, b \rangle, \langle b, c \rangle, \langle c, a \rangle, \langle c, d \rangle, \langle d, c \rangle\}$.

In our method, we are interested in the common substrings of a dataset (either a bad or a good dataset). To this end, we compute sequence sets of individual traces in a dataset and then make a union of all the sequence sets:

$$S_{\Sigma_B,\ell} = \bigcup_{1 \leq i \leq |\Sigma_B|} \{ss_{\sigma_i,\ell} | \sigma_i \in \Sigma_B\} \tag{3.3}$$

$$S_{\Sigma_G,\ell} = \bigcup_{1 \leq i \leq |\Sigma_G|} \{ss_{\rho_i,\ell} | \rho_i \in \Sigma_B\} \tag{3.4}$$

The resulting anomalies is a set of substrings of length ℓ that only occur in the bad dataset, Σ_B, and is computed as follows:

$$anomalies = S_{\Sigma_B,\ell} - S_{\Sigma_G,\ell}, \tag{3.5}$$

where "–" is a set difference.

The length of the substrings, ℓ, is the only parameter in computing the final resulting anomalies. We shall discuss the impact of choosing different values for ℓ in the experimental results section.

Post-processing

After computing anomalies by using (3.5), we get a number of substrings which need to be inspected manually by the user for understanding the cause of the failure. Since each substring is only a part of an explanatory sequence, the programmer needs to identify other parts of the explanatory sequence in the counterexample in order to understand the cause of a failure and subsequently localizing the faulty parts of the model. Therefore, for every substring we also provide to the user a counterexample in which the substring occurs. Since a substring can occur in multiple counterexamples, in order to select one to give to the user, we compare the positions of occurrence of substring in these counterexamples. We choose the counterexample in which the substring occurs earliest (or has the least position of occurrence). Intuitively, we assume that when the substring occurs earlier in a counterexample, it is closer to the beginning of explanatory sequences or the cause of failure, hence the programmer has fewer events to inspect. In the selected counterexample, other distinguishing substrings computed by (3.5) may occur as well, therefore we also provide them to the user.

In order to make the user inspection faster, we propose a *ranking* mechanism for the resulting substrings computed by (3.5). The ranking is based on the position of occurrence of substrings in the corresponding counterexamples. The substrings with smaller position of occurrence in a counterexample are ranked higher. As argued above, it would be easier

for the programmer to locate explanatory sequences using substrings which are ranked higher.

Finally, the output of the method will be a ranked list of tuples of the form:

$$\langle \{a_i \mid a_i \in ss_{\sigma,\ell},\, 0 \leq i \leq |ss_{\sigma,\ell}|\},\, \sigma \rangle,$$

in which $\sigma \in \Sigma_B$ is a counterexample chosen as described above.

Like all other debugging activities in which a *check*, *analyze*, *fix* loop is iterated until all the bugs are fixed, our method should also be used as an iterative process.

Evaluation Score. To evaluate the quality of the outputs generated by our method we propose a quantitative measure that enables us to compare different outputs. We define a score based on the amount of the effort that is required for locating an explanatory sequence in a counterexample by using the output of our method. Since these sequences directly allow the user to identify the faulty part of the model, as we assumed above, the computed score also reflects the amount of manual effort required for locating the faulty part of the model.

The output of our method consists of a number of substrings, so we first define a score for individual substrings. The score of the output will then be the score of the substring which is ranked first in the output. The score of a substring is defined based on the distance in terms of the number of the events between the location of the substring in a counterexample and the first event of an explanatory sequence in the same counterexample. This number, in fact, represents the maximum number of events that the user needs to inspect in the counterexample in order to find an explanatory sequence. As we have explained so far, the identification of explanatory sequences finally needs to be done manually with the aid of the anomalies produced by our method. Therefore, the score of a substring depends on the beginning of the position of the corresponding explanatory sequence in a counterexample. We normalize this distance with respect to the counterexample length.

The following formulas define how a score is computed for a substring produced as the output of our method. Let $Pos(a)_\sigma$ and $Pos(\pi)_\sigma$ denote the position of the substring a and the explanatory sequence π in the counterexample σ, respectively. $|\sigma|$ denotes the length of the counterexample σ. The *score* of the substring a is formally defined as:

$$\begin{aligned} distance &= Pos(a)_\sigma - Pos(\pi)_\sigma \\ score_a &= 1 - \frac{distance}{|\sigma|} \end{aligned} \tag{3.6}$$

For the substring $\langle 31, 32 \rangle$ in our running example (Section 3.1.2), the score is $\frac{30}{72}$, where 30 is according to Equation 3.6 the *distance* between the substring $\langle 31, 32 \rangle$ and the spotted explanatory sequence in the counterexample, and 72 is the length of the counterexample.

If a_0 refers to the first substring in the ranked list of the method output, we define the *output-score* as:

$$output\text{-}score = score_{a_0}, \tag{3.7}$$

where $score_{a_0}$ is the score of a_0 according to (3.6).

3.3 Experimental Results

In this section, we present the result of applying our counterexample explanation method on a number of concurrent systems whose Promela models were taken from [Pel06]. All these models have the deadlock problem. The experiments were performed on a 2.67 GHz PC with 8 GB RAM and Windows 7 64-bit operating system. The experiments were designed to illustrate the effectiveness of the outputs of our method in facilitating understanding the cause of failures in concurrent systems. In the experiments, we show that the anomalies produced by our method can greatly help the programmer in the search for explanatory sequences. We assume that the identification of explanatory sequences (as cause-effect chains) directly allow the user to locate the faulty parts of the model. This assumption is true for all case studies that we used as well as for the example presented in Section 3.1.2.

In Table 3.2, the results of applying our method to six case studies when $\ell = 2$, are given along with the corresponding scores. The name of the corresponding Promela file is given inside the parentheses in front of the name of the model. The average running time of the method for these case studies is 52.44 sec. In this table, the second column shows the number of substrings with length two which are computed by the method according to Equation 3.5. The last column in Table 3.2 represents the number of different explanatory sequences that can be detected by the programmer with the aid of the substrings of length two which are computed by the method. Note that one fault in the code may manifest differently at run time. Therefore, one fault may result in different explanatory sequences. Referring to the method of the generation of the bad dataset in Section 3.2.1, the counterexamples in the bad dataset may be caused due to different faults or defects in the model. Therefore, the substrings computed by our method may give hints to different faults. For example, as Table 3.2 shows for the Brp model, with the aid of the extracted six substrings, we could detect three different explanatory sequences for the occurrence of a deadlock, each implying a different fault in the model. Currently, the programmer has to realize himself/herself whether the extracted substrings imply the same fault or multiple faults.

In Table 3.2, the Brp model has the highest score of 1 which means that the first ranked substring in the output coincides with the start of a sequence that explains a deadlock occurrence. Notice that we use the proposed method as part of an iterative debugging process. After each run of the method, aided by the generated substrings, the user will try to remove as many causes of deadlock as possible. In case the model still contains a deadlock after being modified, the user will apply the method again. This procedure can be iterated until all deadlocks in the model have been removed. As an example, after the

Model	#Substrings ($\ell = 2$)	Score	#Expl. Seq.
Brp(brp.3.pm)	6	1	3
Modified Brp	6	1	2
Rether(rether.4.pm)	24	0.27	15
lann(lann.1.pm)	8	0.97	2
gear(gear.1.pm)	21	0.66	14
train-gate(train-gate.1.pm)	27	0.78	9

#Substrings ($\ell = 2$): No. of Substrings with length two
#Expl. Seq.: No. of Explanatory Sequences

Table 3.2: Summary of the results of the method

first iteration on the Brp model the total number of counterexamples was reduced from 660 to 182 due to the removal of the root cause of some deadlock. The results achieved by applying the method to the modified version of the Brp model in the second iteration are given in the second row of Table 3.2.

Considering the original length of the counterexamples given in Table 3.1, it can be inferred from the data of Table 3.2 the effectiveness of the substrings computed by our method in analysis of counterexamples for understanding the cause of the failure.

By increasing the value of parameter ℓ, the number of the generated substrings will also be increased. Consequently, the programmer needs more effort for examining them. In Table 3.3, the numbers of the generated substrings for $\ell = 2$ and $\ell = 3$ for five case studies are given in the columns "#Substrings $\ell = 2$" and "#Substrings $\ell = 3$", respectively. The last column in this table shows the percentage of increase in the number of the generated substrings. We can see in this table that for the last three case studies, the number of the generated substrings of length three is significantly larger than those with length two. Therefore, the substrings of length three increase the amount of manual effort required for inspecting them. From Table 3.3, we can infer that substrings of length two impose less inspection effort on the programmer when analyzing the counterexamples. As a consequence, the generation of substrings of length three is only done when no substrings of length two can be generated by Equation 3.5 which was not the case for the case studies of Table 3.2.

3.4 Comparison with the Work by Groce and Visser

The most closely related work to ours is that of Groce and Visser [GV03]. It extends Java PathFinder [VM05] with error explanation facilities. Given a counterexample, their method generates a set of *negatives*, which are multiple variations of that counterexample in which the error occurs, and a set of *positives*, which are variations in which the error

Model	#Substrings $\ell = 2$	#Substrings $\ell = 3$	Relative Increase
Brp(brp.3.pm)	6	6	%0
Rether(rether.4.pm)	24	24	%0
lann(lann.1.pm)	8	29	%262.5
gear(gear.1.pm)	21	35	%66
train-gate(train-gate.1.pm)	29	62	%113.8

Table 3.3: Comparison of the number of the substrings with $\ell = 2$ and $\ell = 3$

does not occur. They analyze the common features of each set and the differences between the sets in order to provide an explanation for the counterexample. The focus of their work is on finite counterexamples demonstrating the violation of safety properties such as assertion violations and deadlocks.

To compare our work with theirs, we implemented the algorithm proposed in [GV03] for the generation of a set of positives for a given counterexample inside the SPINJA [dJR10] toolset. The main problem we encountered in applying this algorithm to our case studies was that we could not always generate a non-empty set of positives. This occurred, for instance, in our experiments with the Brp model. Notice that the potential emptiness of the positive set is also mentioned as a potential difficulty in practice in [GV03]. In our method, on the other hand, we consider the complete set of good traces that can be generated with the aid of SPIN, and hence we cannot encounter the problem of an empty positive set for any case study that does at all reveal a "good" behavior.

The work in [GV03] proposes three different analyses for explaining counterexamples, namely *transition* analysis, *invariant* analysis and minimal *transformation* analysis between negatives and positives. Among these three analyses, only the third one, which takes the order of execution of actions into account, is similar to our method and can be used for revealing concurrency problems such as unforeseen interleavings. In this analysis, the authors of [GV03] compare a negative and a positive in order to determine the divergent sections of what they refer to as a state-action path. These divergent sections along with the associated positive and negative form a *transformation*. In Figure 3.5, a negative with 64 events along with a positive with 473 events derived for the Rether case study [Pel06], are given. Due to space limitations, only the first 20 events and the last 15 events of these traces are shown in this figure. The events in this figure are represented as a combination of two numbers separated by a " . ". The number at the left side of " . " corresponds to the event id and the number at the right side of " . " corresponds to the proctype id. For example, "413.11" refers to the event with $\mathsf{id} = 413$ which is issued by proctype 11. The first 19 events are identical both in the positive and in the negative, thus the divergent sections start from event 20 in both traces. These divergent sections last until the end of the positive and the negative since they do not share a common portion at the end of their traces. Therefore, the transformation generated

by [GV03] will consist of two traces with 45 and 454 events. However, in our method two substrings of length two, $\langle 369.10, 375.10\rangle$ and $\langle 375.10, 9.1\rangle$, as well as the negative itself with 64 events are given to the programmer for further analysis. We conclude that while with the transformation analysis of [GV03] the programmer needs to inspect traces of 45 and 454 events, in our method he/she needs to inspect at most 48 events in order to understand how a deadlock occurs. 48 is, in fact, the number of events between the position of $\langle 369.10, 375.10\rangle$ and the position of event "2.1" which is the beginning of the explanatory sequence for the occurrence of a deadlock in the counterexamples. These two positions in the counterexample are 62 and 15, respectively, and in Figure 3.5 they are connected by arrows and straight lines. In conclusion, our method appears, at least for the case study we considered here, to require less effort on behalf of the programmer in order to understand the reason for the failure than the equivalent analysis according to the work in [GV03].

3.4.1 Related Work

There are a number of works on automatically explaining counterexamples using different technical approaches and having different objectives. The work documented in [BBDC+09] using the notion of *causality* introduced by Halpern and Pearl [HP05] formally defines a set of causes for the failure of a property on a given counterexample trace (See Chapter 2, Section 2.1.3). For the explanation of a counterexample, this method deals with what values on the counterexample cause it to falsify the property. In [WYIG06] Wang et al. focus on explaining the class of assertion violation failures. Their method uses an efficient weakest precondition algorithm which is executed on a single concrete counterexample in order to extract a minimal set of contradicting word-level predicates. Groce et al. [GCKS06] developed a tool called Explain, which extends the CBMC model checker [KCL04], for assisting users in understanding and isolating errors in ANSI C programs based on Lewis' counterfactual causality reasoning(See Chapter 2, Section 2.1.3). Given a counterexample, Explain finds the most similar successful execution based on a *distance metric* on execution traces. The differences (Δs) between the most successful execution and the counterexample, after being refined by a slicing step, is given to the programmer as an explanation. The distance between executions a and b is measured based on the number of the variables to which a and b assign different values. In contrast to the three methods cited above, our counterexample analysis method does not consider any values that are assigned to variables, instead only the order of execution of actions inside execution traces are taken into account. Therefore, we are able to give explanations to counterexamples in which the violation of a property is due to a specific *order* of execution of actions. Moreover, the other methods are based on an analysis of one single counterexample while in our method for extracting commonalities we use non-singleton sets of counterexamples.

The work by Ball et al. [BNR03] compares a counterexample with a set of similar correct traces in order to extract single program statements that are only executed in the counterexample. These program statements are reported to the user as the suspicious

parts of the program code that are likely to be the cause of the violation of the property. In this method, if a counterexample violates a property at some control location c of the program code, then the execution traces that reach to c without violating the property are considered as similar correct traces. The method has been implemented in the context of the SLAM project in which a software model checker that automatically verifies temporal safety properties of C programs has been developed [BR02]. Since this method only considers single program statements, it cannot express counterexamples in which the violation of a property is due to a specific *order* of execution of actions. The criteria they use for finding similar correct traces are similar to those used by the method in [GV03]. In fact, the method in [GV03] is most closely related to ours, so we provide a detailed comparison of this method with ours at the beginning of this section.

There are a few fault localization techniques based on testing which are analogous to ours and consider the actual order of execution of the statements in the program in order to locate the fault in the program code [NAW+08] [DLZ05]. The work of [DLZ05] had an important influence on our method.

3.5 Summary

In this chapter, we presented an anomaly detection method for explaining the model checking counterexamples demonstrating the violation of a desired property in message passing concurrent system models. In particular, we have focused on deadlock detection using the SPIN model checker. By comparing a set of counterexamples with a set of correct traces that never deadlock, we extract a number of explanatory sequences that prove to point to the root cause of the deadlock occurrence in the model. Experimental results showed the effectiveness of our method and discussed measures to reduce the effort of the model designer when localizing the root cause for the occurrence of a deadlock in the model. We also compared our work extensively to related work, in particular the approach by Groce and Visser.

```
1  active proctype Bandwidth() {
2  byte i = 0;

4  idle: if
5  :: reserve ? i; goto res;
6  :: release ? i; goto rel;  (2)
7  fi;

9  rel: if
10 :: atomic {in_RT[i] == 1;  (4)
11           ok ! 0;  (5)
12           in_RT[i] = 0;  (33)
13           RT_count = RT_count-1;}
14           goto idle;

16 :: in_RT[i] == 0; goto error_st;
17 fi;

19 ...

21 error_st:
22 false;
23 }
```

```
1  active proctype Token() {
2  byte i = 0;
3  ...
4  start: if
5  :: i = 0; goto RT_phase;
6  fi;

8  RT_phase: if
9  ::  d_step {i < 2 && in_RT[i] == 0;
10             i = i + 1;} goto RT_phase;

12 ::  atomic {i == 0 && in_RT[i] == 1; (28)
13             visit_0 ! 1;}  (29)
14             goto RT_wait;

16 ::  atomic {i == 1 && in_RT[i] == 1;
17             visit_1 ! 1;} goto RT_wait;

19 :: i == 2; goto NRT_phase;
20 fi;

22 RT_wait: if
23 ::  atomic {done ? 0; i = i + 1;}
24     goto RT_phase;

26 fi;
27 ...
28 }
```

```
1  active proctype Node_0() {
2  byte rt = 0;
3  byte granted = 0;

5  idle: if
6  :: visit_0 ? rt; goto start;  (30)
7  fi;

9  start: if
10 :: rt == 1; goto RT_action;  (31)
11 :: rt == 0; goto NRT_action;
12 fi;

14 RT_action: if
15 :: granted == 0; goto error_st; (32)
16 :: goto finish;
17 :: atomic {release ! 0; (1)
18       granted = 0;} (3)
19    goto wait_ok;
20 fi;

22 ...

24 finish: if
25 :: done ! 0; goto idle;
26 fi;

28 wait_ok: if
29 :: ok ? 0; goto finish;  (6)
30 fi;

32 error_st:
33 false;
34 }
```

Figure 3.3: Parts of the Promela model of the Rether Protocol

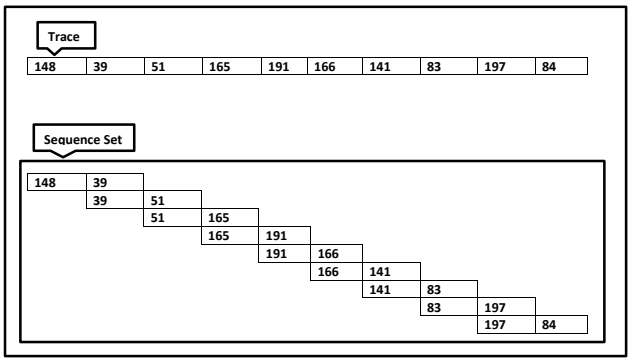


Figure 3.4: A trace fragment along with a sequence set of length two

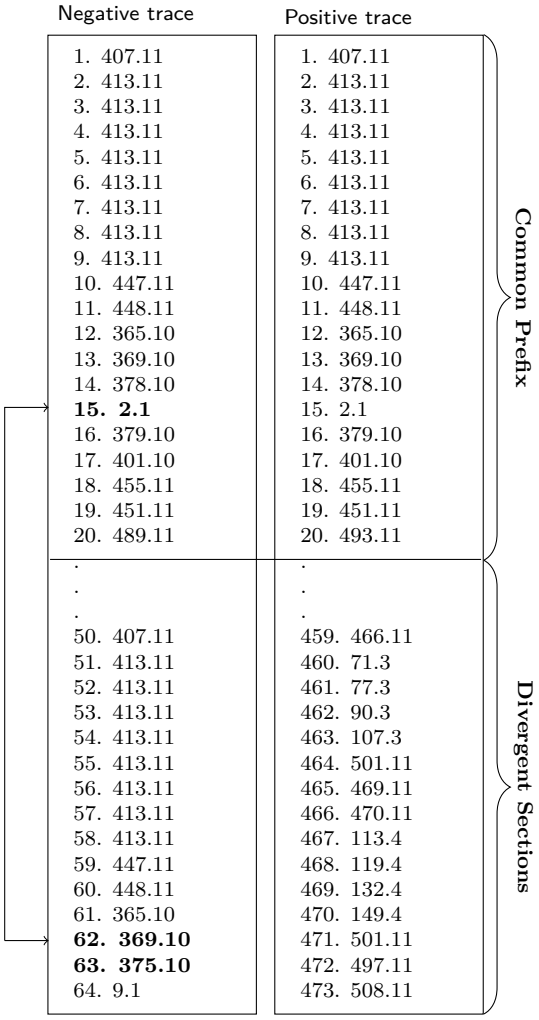


Figure 3.5: A *negative* and a *positive* trace for the Rether protocol

CHAPTER 4

Mining Sequential Patterns to Explain Concurrent Counterexamples

In Chapter 3, we proposed an anomaly detection method for explaining concurrent counterexamples. We showed that using our method, concurrency bugs can be explained in general by only analyzing the good and the bad traces and without exploiting the characteristics of specific bugs such as deadlocks or atomicity violations. For the method of Chapter 3, we introduced the concept of *explanatory sequences* as an ordered sequence of events inside counterexamples which reveal the specific orders between concurrent events inside a counterexample which are presumed to be causal for the property violation. We showed that due to scheduling the events belonging to an explanatory sequence do not occur necessarily next to each other. In general, they occur as a *subsequence* (Definition 5) of a counterexample showing that events belonging to an explanatory sequence can occur at an arbitrary distance from each other. Therefore, we maintained that *sequential pattern mining* algorithms, which extract the frequent subsequences from a dataset of sequences without limitations on the relative distance of events belonging to the subsequences, are an adequate and obvious choice to extract explanatory sequences from large sets of counterexamples. However, as we discussed in Chapter 3, Section 3.1.4, mining sequential patterns from the datasets of counterexamples generated from typical concurrent system models is intractable. To address the complexity challenges we encountered in mining sequential patterns from the bad and the good datasets, we proposed as an approximation to extract substrings comprising consecutive events, from the traces of bad and good datasets for counterexample explanation.

In this chapter, we improve our explanation by extracting sequences of events which do not necessarily occur contiguously inside counterexamples. In other words, we propose a method for extracting explanatory sequences which occur as subsequences of

counterexamples. The method uses sequential pattern mining techniques for extracting explanatory sequences. However, to make the mining problem more tractable, we propose a trace length reduction technique.

4.1 Preliminaries

Since the setting in this work is similar to the work in Chapter 3, we refer the reader to Section 3.1 of that chapter for definition of counterexamples, system models, traces, and properties. In this section, we present a running example which will be used for describing the method. Moreover we provide more details on the functioning of sequential pattern mining techniques since they are used for computing anomalies in the method of this chapter.

4.1.1 Running Example

Using this example, similar to the motivating example of Chapter 3, Section 3.1.2, we illustrate how a deadlock can occur due to the temporal order of execution of a set of actions in the model of a concurrent system. Referring to this example, we motivate that *contrasting* sequential patterns of the bad and good datasets can reveal the ordered sequences of actions that can help to explain the violation of a property, such as a deadlock in a concurrent system. We use the model of a preliminary design of a plain old telephony system (POTS)[1] as an example. This model was generated with the visual modeling tool Vip [KL00] and contains a number of deadlock problems. It comprises four concurrently executing processes corresponding to two users and two phone handlers. Each user in this model talks to a phone handler for making calls. The phone handlers are communicating with each other in order to switch and route user calls.

In the Promela code of the POTS case study, there exist four proctypes, namely User1, PhoneHandler1, User2, and PhoneHandler2. Every user communicates with one phone handler via two channels with capacity one. One of the two channels between a user and a phone handler is used for sending messages from the user to the corresponding phone handler and the other one is used for the opposite direction (Figure 4.1). The communication between users and corresponding phone handlers is *asynchronous* since channels have capacity greater than zero in the Promela code [Hol03] [2]. A proctype blocks when it tries to send a message via a full channel.

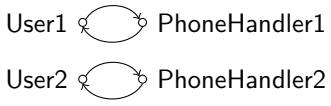
In Figure 4.2, a fragment of a counterexample indicating the occurrence of a deadlock in the POTS model, is given. In this figure, vertical bars are used for showing the relative order between the execution of the instructions from different proctypes (time increases in these bars from top to bottom). In this figure, arrows depict "send" and

[1]The Promela code of the POTS case study is available at http://www.inf.uni-konstanz.de/soft/tools/CEMiner/POTS7-mod-07-dldetect-never.prm. (July 2012)

[2]In Promela, channels with capacity zero are used for *synchronous* communications like the channels in the Rether protocol, the running example of Chapter 3.

"receive" instructions, and are labeled with messages (dashed arrows correspond to receive instructions).

Figure 4.1: POTS channels



From the sequence of send and receive messages in Figure 4.2, it can be inferred that the channel from PhoneHandler1 to User1 becomes full after sending the dial-tone message for the second time because there exists no corresponding receiving message from User1. Therefore, PhoneHandler1 blocks after executing the statement phone_number != 1 while trying to send a busy-tone message to User1. User1 also blocks after sending the second on-hook which makes the channel from User1 to PhoneHandler1 also full. Due to the symmetry in the model, a similar interaction (which has not be shown in Figure 4.2) also occurs between the User2 and PhoneHandler2 which finally leads the system to a deadlock state. The presumed assumption of the model designer is that a user is synchronized with the corresponding phone handler so that when the phone handler sends a dial-tone message, the user subsequently receives it before taking any other action. However, as we can see in Figure 4.2, the model is faulty, therefore, the first dial-tone messages is only received after six events, and the second one has never been received.

In order to realize the problem in Figure 4.2 which we described above, the model designer needs to inspect all the events issued by User1 and PhoneHandler1. This is because the order between the sending and receiving messages reveal how these two processes block. In Figure 4.2, by projecting the trace into events of User1 and PhoneHandler1, we get a sequence of 14 events in which the events do not occur adjacently inside the counterexample. Instead, they are interspersed with unrelated events belonging to the interaction of the User2 and PhoneHandler2.

As we discussed in Chapter 3, in general events belonging to an explanatory sequence can occur at an arbitrary distance from each other due to the non-deterministic scheduling of concurrent events. In other words, they occur as subsequences (Definition 5) of the counterexamples.

4.1.2 Sequential Pattern Mining (revisited)

In Section 3.1.3 of Chapter 3, we briefly introduced the terminology used in sequential pattern mining algorithm. Since in the method of this chapter we use this type of algorithm for detecting anomalies we provide more details on it such as defining *closed* patterns. The following paragraph is repeated from the previous chapter in order to make the presentation self-contained.

A sequence dataset S, $\{s_1, s_2, ..., s_n\}$, is a set of sequences. The support of a sequence p in S is the number of the sequences in S that p is a subsequence of: $\mathsf{support}_S(p) = |\{s \mid s \in S \wedge p \sqsubseteq s\}|$. Given a minimum support threshold, min_supp, the sequence p is considered a *sequential pattern* or a *frequent subsequence* if its support is no less than min_supp: $\mathsf{support}_S(p) \geq \mathsf{min\_supp}$. We denote the set of all sequential patterns mined

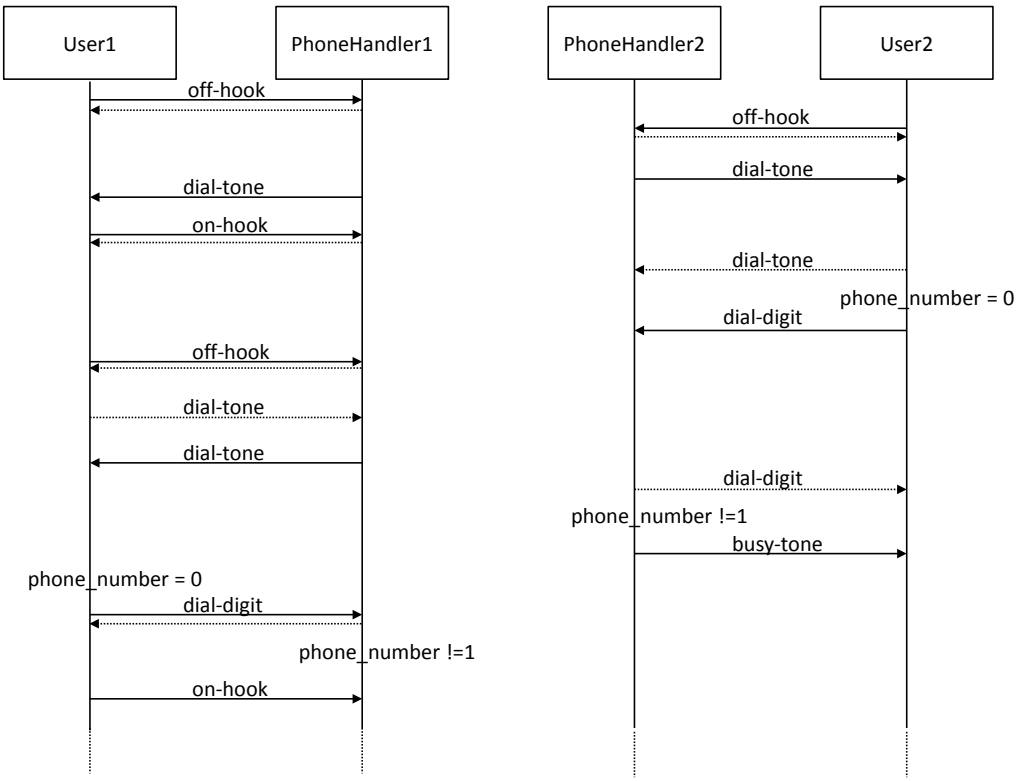


Figure 4.2: A fragment of a counterexample in POTS model

from S with the given support threshold $\mathsf{min\_supp}$ by $\mathrm{FS}_{S,\mathsf{min\_supp}}$, i.e., $\mathrm{FS}_{S,\mathsf{min\_supp}} = \{p \mid \mathsf{support}_S(p) \geq \mathsf{min\_supp}\}$.

As an example, consider a sequence dataset S that has five sequences:

$$\begin{aligned} S \;=\; & \{\langle \mathsf{a},\mathsf{b},\mathsf{c},\mathsf{e},\mathsf{d}\rangle, \\ & \langle \mathsf{a},\mathsf{b},\mathsf{e},\mathsf{c},\mathsf{f}\rangle, \\ & \langle \mathsf{a},\mathsf{g},\mathsf{b},\mathsf{c},\mathsf{h}\rangle, \\ & \langle \mathsf{a},\mathsf{b},\mathsf{i},\mathsf{j},\mathsf{c}\rangle, \\ & \langle \mathsf{a},\mathsf{k},\mathsf{l},\mathsf{c}\rangle\} \end{aligned}$$

With $\mathsf{min\_supp} = 4$, we obtain:

$$\begin{aligned} \mathrm{FS}_{S,4} \;=\; & \{\langle \mathsf{a}\rangle : 5, \\ & \langle \mathsf{c}\rangle : 5, \\ & \langle \mathsf{a},\mathsf{c}\rangle : 5, \\ & \langle \mathsf{b}\rangle : 4, \\ & \langle \mathsf{a},\mathsf{b}\rangle : 4, \\ & \langle \mathsf{b},\mathsf{c}\rangle : 4, \\ & \langle \mathsf{a},\mathsf{b},\mathsf{c}\rangle : 4\} \end{aligned}$$

where the numbers following the patterns denote the respective supports of the patterns.

Since mining all sequential patterns will typically result in a combinatorial number of patterns, algorithms, such as [YHA03, WH04] only mine *closed* sequential patterns. A closed sequential pattern is a pattern which does not have any *super-sequence* (Definition 5 in Chapter 3) with the same support. In $\mathrm{FS}_{S,4}$, $\langle \mathsf{a}, \mathsf{b}, \mathsf{c} \rangle$ is a closed pattern while $\langle \mathsf{b}, \mathsf{c} \rangle$ and $\langle \mathsf{a}, \mathsf{b} \rangle$ are not closed since the $\mathsf{support}$ of their super-sequence $\langle \mathsf{a}, \mathsf{b}, \mathsf{c} \rangle$ is equal to their $\mathsf{support}$s (which is 4). A closed pattern encompasses all the frequent patterns with the same support value which are all subsequences of it. For example, in $\mathrm{FS}_{S,4}$, the closed pattern $\langle \mathsf{a}, \mathsf{b}, \mathsf{c} \rangle$ encompasses three patterns including $\langle \mathsf{b} \rangle$, $\langle \mathsf{a}, \mathsf{b} \rangle$, and $\langle \mathsf{b}, \mathsf{c} \rangle$ (but not $\langle \mathsf{a}, \mathsf{c} \rangle$ since its $\mathsf{support}$ is 5).

The set of all closed sequential patterns mined from S with the given support threshold $\mathsf{min\_supp}$, denoted by $\mathrm{CS}_{S,\mathsf{min\_supp}}$, is defined as follows:

Definition 8. *$CS_{S,\mathsf{min\_supp}} = \{\pi \mid \pi \in FS_{S,\mathsf{min\_supp}} \wedge \nexists \pi' \in FS_{S,\mathsf{min\_supp}}$ such that $\pi \sqsubset \pi' \wedge \mathsf{support}(\pi) = \mathsf{support}(\pi')\}$.*

Since every frequent pattern is represented by a closed pattern, mining closed patterns leads to a more compact result set. For example, while $\mathrm{FS}_{S,4}$ contains seven patterns, $\mathrm{CS}_{S,4}$ has only two patterns:

$$\begin{aligned} \mathrm{CS}_{S,4} \quad = \quad & \{\langle \mathsf{a}, \mathsf{c} \rangle : 5, \\ & \langle \mathsf{a}, \mathsf{b}, \mathsf{c} \rangle : 4\} \end{aligned}$$

4.2 Counterexample Explanation Method

4.2.1 Computing Anomalies

In our method we mine only closed patterns in order to avoid a combinatorial explosion. For explaining counterexamples, we first mine closed sequential patterns from the bad (Σ_B) and the good datasets (Σ_G) with the given minimum support thresholds $\mathsf{min\_supp}_B$ and $\mathsf{min\_supp}_G$, respectively. We refer to the sets of closed patterns mined from the bad and the good datasets as $\mathrm{CS}_{\Sigma_B,\mathsf{min\_supp}_B}$ and $\mathrm{CS}_{\Sigma_G,\mathsf{min\_supp}_G}$, respectively. Contrasting the sequential patterns of the good and the bad datasets results in *anomalies* which are the patterns that are only frequent in the bad dataset and is defined formally as:

Definition 9. *$AS_{\mathsf{min\_supp}_B,\mathsf{min\_supp}_G} = \{\pi \mid \pi \in CS_{\Sigma_B,\mathsf{min\_supp}_B} \wedge \pi \notin CS_{\Sigma_G,\mathsf{min\_supp}_G}\}$, where $AS_{\mathsf{min\_supp}_B,\mathsf{min\_supp}_G}$ denotes the set of anomalies that can be extracted from Σ_B and Σ_G by mining closed patterns using minimum support thresholds $\mathsf{min\_supp}_B$ and $\mathsf{min\_supp}_G$.*

The set $\mathrm{AS}_{\mathsf{min\_supp}_B,\mathsf{min\_supp}_G}$ is computed according to (4.1):

$$\mathrm{AS}_{\mathsf{min\_supp}_B,\mathsf{min\_supp}_G} = \mathrm{CS}_{\Sigma_B,\mathsf{min\_supp}_B} - \mathrm{CS}_{\Sigma_G,\mathsf{min\_supp}_G}, \tag{4.1}$$

where "–" denotes a set difference operation.

The anomalies computed according to (4.1) are a set of ordered sequences of events which give an explanation for the property violation. As we will show in the experiments, the extracted set of anomalies is indicative of one or several faults inside the model. These anomalies can hence be used as the clues to the exact location of the faults inside the model and thereby greatly facilitate the manual fault localization process.

4.2.2 Trace Length Reduction

In Section 3.1.4 of Chapter 3 we discuss that mining sequential patterns from the datasets of counterexamples is intractable. As we argue, this observation is due to inherent characteristics of those datasets, in particular the average length of the sequences and the number of different events that they include. We conclude that we need some technique for reducing the length of the counterexamples in order to make the use of sequential pattern mining in this application domain tractable.

To reduce the length of the traces, we propose to partition the traces into *subtrace*s which are significantly shorter than original traces, and consequently mining patterns from them becomes feasible. To this end, we exploit the *recurrence* of events inside the traces of non-terminating communication protocols which we are mainly focusing on. The system models of these message passing concurrent systems contain only a *few* number of basic statements (see Section 3.1.1 of Chapter 3). Therefore, in the (supposedly infinite) traces of these models events occur repeatedly (infinitely often).

In the model of a communication protocol, we have a number of processes (proctypes in Promela) that communicate via message channels in order to achieve a common goal. Therefore, the communications in this model are not random, but according to *scenarios* which are defined at design time. Scenarios which are illustrated typically using *sequence diagrams* [3] specify how processes and in what order they interact. For example, Figure 4.3 depicts two scenarios for the POTS model. The vertical bars (referred to as *lifelines* in a sequence diagram) represent the temporal order between the sending and receiving of different messages. Obviously, there are more scenarios in the POTS model such as interactions between two phone handlers, however Figure 4.3 only shows two simple scenarios involving only one user and one phone handler.

A trace of a communication protocol comprises instances of its scenarios. For example, Figure 4.4 illustrates a trace of the POTS model consisting of one instance of scenario 1 in Figure 4.3 between User1 and PhoneHandler1 and two instances of scenario 2 one between User1 and PhoneHandler1 and the other between User2 and PhoneHandler2. Note that the events belonging to interactions of User1 and PhoneHandler1 are interspersed with interactions between User2 and PhoneHandler2. This is because scenarios in Figure 4.3 define only interactions between one user and one phone handler. Therefore, two pairs of one user and one phone handler can interact concurrently.

[3] https://en.wikipedia.org/wiki/Sequence_diagram.

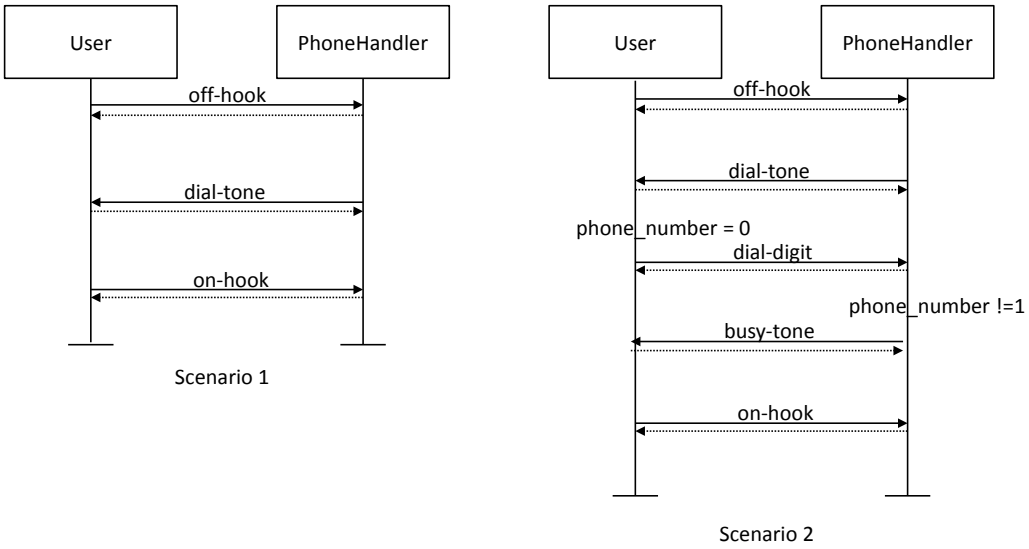


Figure 4.3: Two simple scenarios for the POTS model

As we have seen above, a trace of a communication protocol can be partitioned into fragments, each corresponding to an instance of a scenario. This observation lead us to the idea of partitioning the traces into subtraces for reducing the length of the original traces. Ideally, each subtrace corresponds to an instance of a scenario. Instead of mining original traces, subtraces are then taken into account.

Partitioning the Traces into Subtraces

In order to partition a trace into subtraces that each corresponds to an instance of a scenario, we need to break the trace at the occurrence of an initiating event of a scenario. For example, in the Rether protocol (Section 3.1.2 of Chapter 3) all the scenarios start with the execution of statement $\mathsf{i} = 0$ in Token proctype which is Line 5 in Figure 3.3. Therefore, traces can be decomposed into subtraces by breaking them at the events corresponding to the execution of this statement. In the POTS model, there exist two initiating events: sending an off-hook message from User1 to PhoneHandler1 or sending the same message from User2 to PhoneHandler2. As we will see in the experimental results section, for this case study we broke the traces at sending an off-hook message from User1 to PhoneHandler1 because this event occurs more frequently in the traces than the other one, and moreover the resulting subtraces are shorter. Table 4.1, shows the number of subtraces and their average lengths by breaking the traces in Σ_B at the occurrence of these two events. Note that the initiating events are given as input to the method, and are not currently determined automatically (cf. Figure 4.6).

For protocols, like POTS, where there exist more than one initiating event for scenarios, determining an event at which to break up the traces is a heuristic decision. One strategy

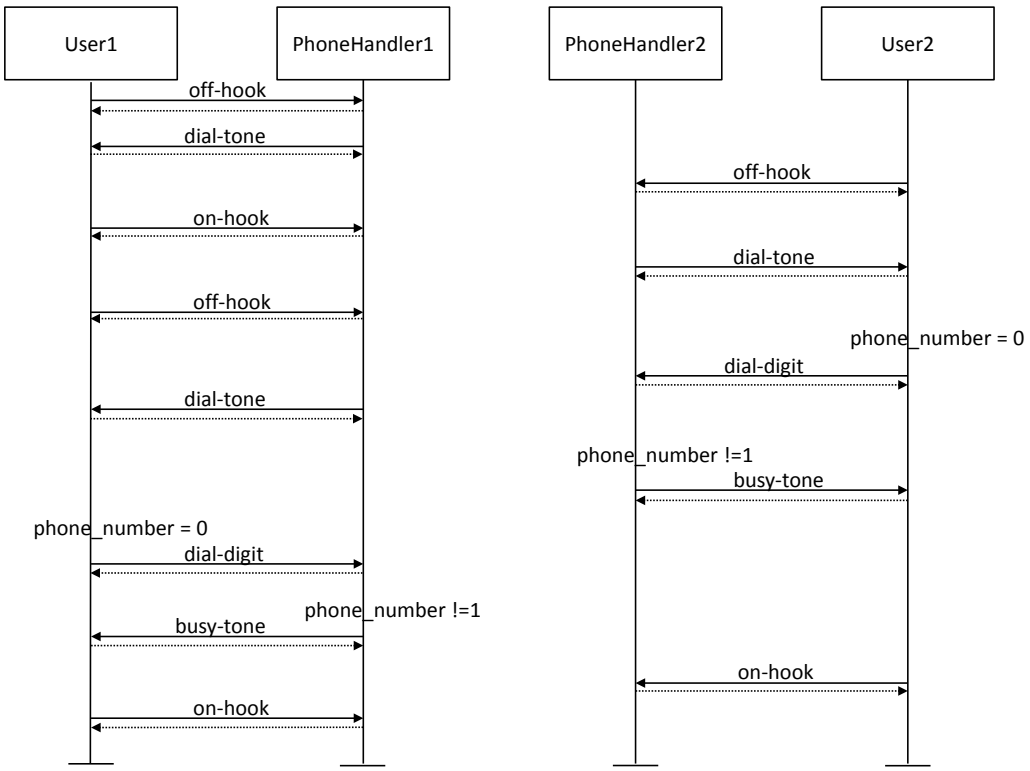


Figure 4.4: A sample trace consisting of three scenarios for the POTS model

Event: off-hook	#Subtraces	Subtraces Avg. Len.	Avg. Len. Red. (%)
User1 → PhoneHandler1	2046	25	98.5%
User2 → PhoneHandler2	1444	36	97.8 %

Table 4.1: Results of breaking the traces in Σ_B of the POTS model at two initiating events.

for determining the event to break up the traces is by calculating how much reduction can be gained on the average from each individual action, and then to choose the one with the highest reduction ratio. Another heuristic is choosing the event which divide the traces more evenly or result in subtraces with more similar length. In our experiments, we chose the event which is more frequent in the bad dataset, and moreover results in a higher reduction ratio, for instance, for the POTS model we chose sending an off-hook message from User1 to PhoneHandler1 (Table 4.1).

Let r be an initiating event for scenarios inside a trace $\sigma = \langle e_1, e_2, ..., e_n \rangle$. If r occurs m times inside σ, breaking σ at the m positions of r in σ results in $m+1$ subtraces (or m if σ starts with r or ends with it). Each subtrace contains the sequence of events between each two consecutive positions of r in σ:

Definition 10. *Let integers* $0 \leq r_1 < r_2 < ... < r_m \leq n$ *refer to* m *positions of* r *inside* σ. $\sigma_{i,r} = \langle e_{r_{i-1}}, e_{r_{i-1}+1}, ..., e_{r_i-1} \rangle$ *where* $1 < i < m+1$ *is the* i^{th} *subtrace of* σ. *The first* $\sigma_{1,r}$ *and the last* $\sigma_{m+1,r}$ *subtrace are* $\langle e_1, ..., e_{r_1-1} \rangle$ *and* $\langle e_{r_m}, ..., e_n \rangle$, *respectively.*

Instead of analyzing the temporal order between all the events of σ, we examine the temporal order between the events of a subtrace $\sigma_{i,r}$, $1 \leq i \leq m+1$ in isolation. Since each subtrace provides a local view on the sequence of events occurred inside σ, by examining the temporal order between the events of a subtrace, we, therefore, lose a global view of the trace. However, due to our decomposition technique which tries to map the local views to individual scenarios, we are still able to isolate problematic interactions inside a single scenario. For example, for the counterexample given in Figure 4.2 if we divide the trace when User1 goes off-hook, we get two subtraces as given in Figure 4.5. Either of these two subtraces reflect the synchronization problem in the model: a dial-tone message is sent by PhoneHandler1, however User1 subsequently does not receive it before taking any other action. Notice that as a consequence of this abstraction we lose access

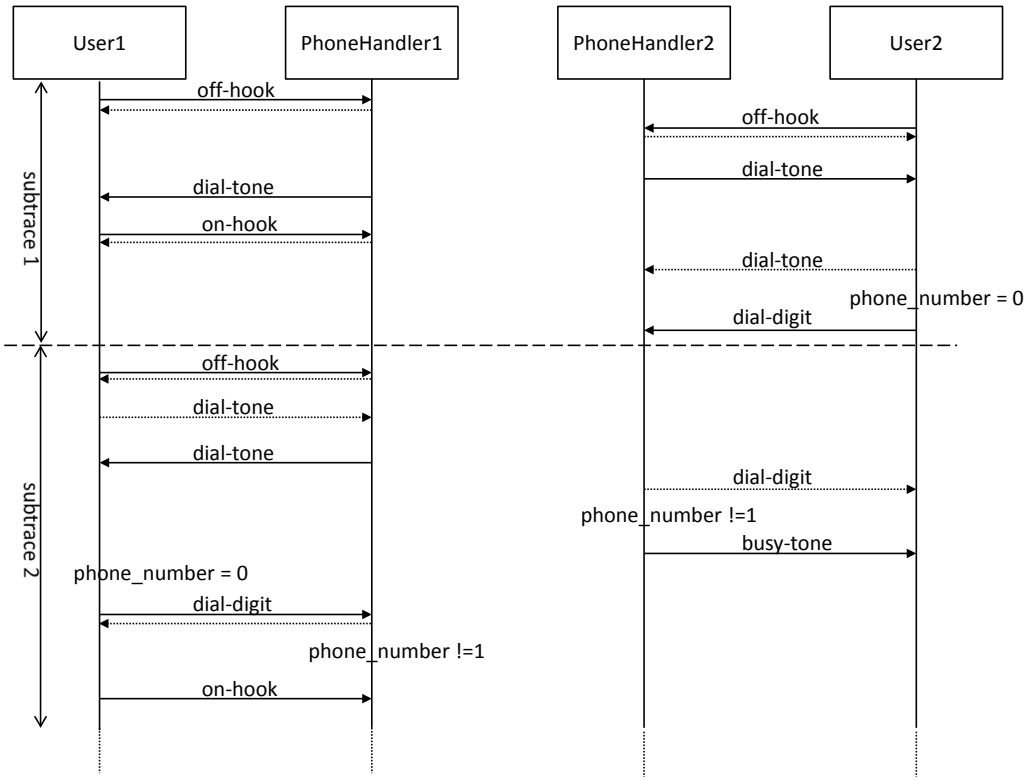


Figure 4.5: A counterexample in POTS model divided into two subtraces

to the causes of failures that spread over multiple scenarios.

As we will see in the experimental results section, this reduction technique can reduce the average sequence length of the datasets significantly, and hence can make mining sequential patterns from them feasible.

Threats to Validity.

It should be noted that this reduction technique is mainly applicable to traces comprising scenarios which specify how the concurrent components should interact, such as non-terminating communication protocols. For some large models the proposed reduction technique may still not sufficiently reduce the length of the traces. As we have seen the produced anomalies for explaining counterexamples only contain one instance of the initiating event r for scenarios. If however for understanding the cause of the property violation inside the counterexample, the isolation of an ordered sequence of events containing more than one instance of r is required, then the analysis of the subtraces would not be sufficient. In other words, since we lose some temporal order by analyzing only the subtraces, there may exist some concurrency bugs which our method cannot explain.

4.2.3 Contrasting Sequential Patterns

Instead of mining patterns from the bad Σ_B and good Σ_G datasets, we mine patterns from the datasets of subtraces obtained by breaking the traces in original datasets at the occurrence of some given event, for instance, r. We refer to the resulting datasets which contain the subtraces of Σ_B and Σ_G as $subt(\Sigma_B, r) = \{\sigma_r \mid \sigma_r$ is a subtrace of σ and $\sigma \in \Sigma_B\}$ and $subt(\Sigma_G, r) = \{\phi_r \mid \phi_r$ is a subtrace of ϕ and $\phi \in \Sigma_G\}$, respectively.

In analogy with Equation 4.1, anomalies are then computed by

$$\mathrm{AS}_{\mathsf{min\_supp}_B,\mathsf{min\_supp}_G,r} = \mathrm{CS}_{subt(\Sigma_B,r),\mathsf{min\_supp}_B} - \mathrm{CS}_{subt(\Sigma_G,r),\mathsf{min\_supp}_G}. \tag{4.2}$$

For mining closed sequential patterns we use an algorithm called CloSpan [YHA03]. The flowchart of our method is given in Figure 4.6.

The anomalies generated by Equation 4.2 are the distinguishing patterns representing the set of sequences of actions that are only frequent or typical in the bad dataset. The user defined threshold values, $\mathsf{min\_supp}_B$ and $\mathsf{min\_supp}_G$ in Equation 4.2 are, in fact, the parameters of our method. By decreasing the value of the support threshold, the number of the generated sequential patterns from a dataset of traces increases.

Post-processing. In order to reduce the number of the mined patterns, we remove the patterns which are substrings of some other generated pattern. This is because the ordering relationship that can be inferred from these patterns can also be inferred from the longer patterns that these patterns are substrings of (Filtering Patterns step in Figure 4.6).

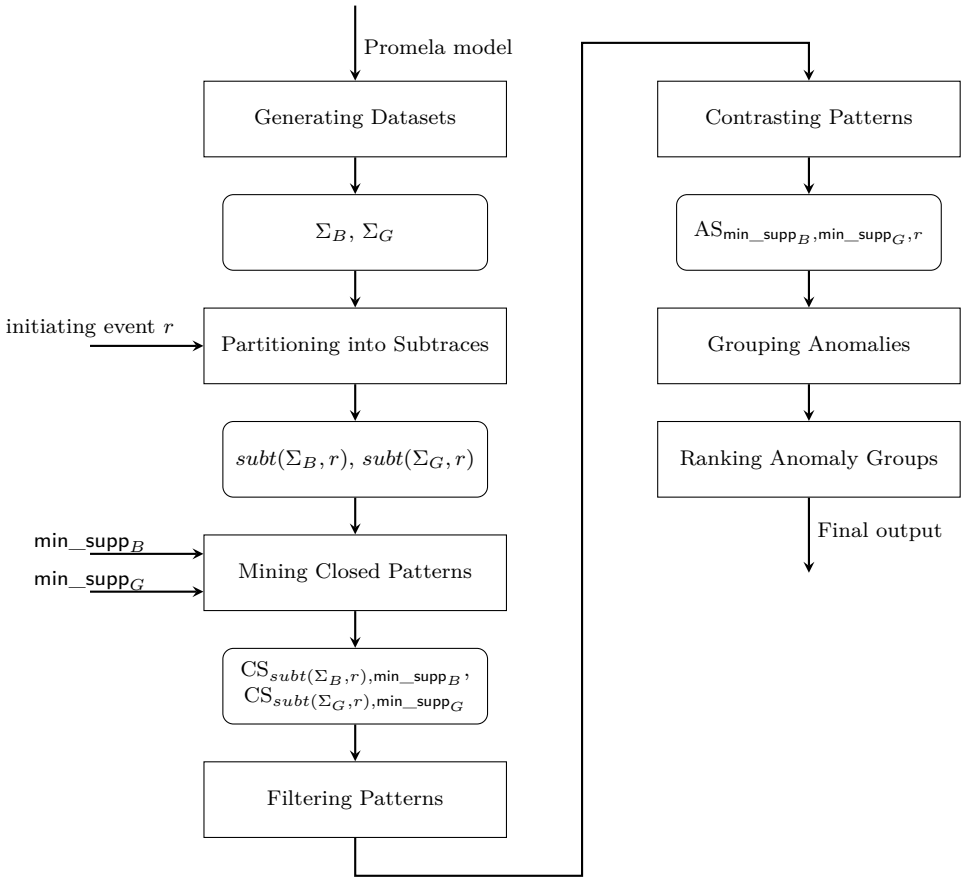


Figure 4.6: Flowchart of the counterexample explanation method

Moreover, to facilitate the interpretation of the result set obtained by Equation 4.2 we divide the anomalies into a number of groups so that each group contains patterns which are all subsequences of the longest pattern in that group. Intuitively, these patterns refer to the same problematic interaction. Figure 4.7 shows an example of such a group of patterns. In this figure, numbers refer to event ids (Definition 3). The longest pattern containing 9 events and appearing at the top, is the super-sequence of the other two patterns. Inspecting the patterns of one group reveal interesting temporal orders between the events. For example, one temporal order that can be inferred from the longest pattern in Figure 4.7 is between three events with ids "334", "1426", and "444": $\langle 334, 1426, 444 \rangle$. From the other two patterns which are subsequences of the longest pattern, it can be inferred that not always event "1426" occurs between events "334" and "444" because the sequence $\langle 334, 444 \rangle$ is also frequent, and not always event "1426" is preceded by event "334" because the sequence $\langle 1406, 1426, 444 \rangle$ is also frequent.

The groups of patterns are then ordered based on the length of the longest pattern inside them. Groups with the shorter length of the longest pattern will be ranked higher because the analysis of these patterns by the user requires less effort.

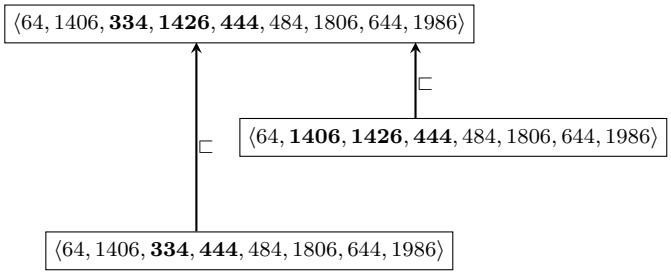


Figure 4.7: Patterns inside one group

4.3 Experimental Evaluation

The experiments that we report on in this section were performed on a 2.67 GHz PC with 8 GB RAM and Windows 7 64-bit operating system. The prototype implementation of our method was realized using the programming language C#.Net 2010. We discuss the results obtained by applying our method to a number of case studies. In these experiments, we evaluated our method from two main aspects:

- Can the proposed length reduction technique *efficiently* shorten the traces, such that mining sequential patterns from the traces becomes feasible?
- How *effectively* can the generated anomalies reveal the problematic interactions in concurrent system, hence explaining concurrency bugs?

Case Study 1: POTS Model. We first applied our method to the POTS model (see Section 4.1.1) in order to obtain explanations for the occurrence of deadlocks. The traces were shortened in length by breaking them at the positions User1 sends an off-hook message to PhoneHandler1 as it has been discussed in Section 4.2.2. In order to study the effect of the threshold value (min_supp) on the number of the generated patterns appearing in the final result set, we applied different threshold values (min_supp), starting with a comparatively high threshold value of 90%. Figure 4.8 illustrates how fast the number of the computed closed sequential patterns in the datasets increases by decreasing the threshold value (min_supp). To understand the cause of the failure, the patterns in the final result set needs to be inspected by the model designer. Inspection of a result set with fewer number of patterns requires less effort. Therefore, we gradually decrease the thresholds until explanatory sequences emerge. Moreover, Figure 4.8 shows the effectiveness of our filtering step in reducing the number of the generated patterns. The reduction is by a factor of approximately 0.5.

Table 4.2 shows the amount of the length reduction we gained by applying our length reduction technique on the bad and good datasets of the POTS model. From this table, it can be inferred partitioning the traces into subtraces is quite efficient in reducing the length of the original traces.

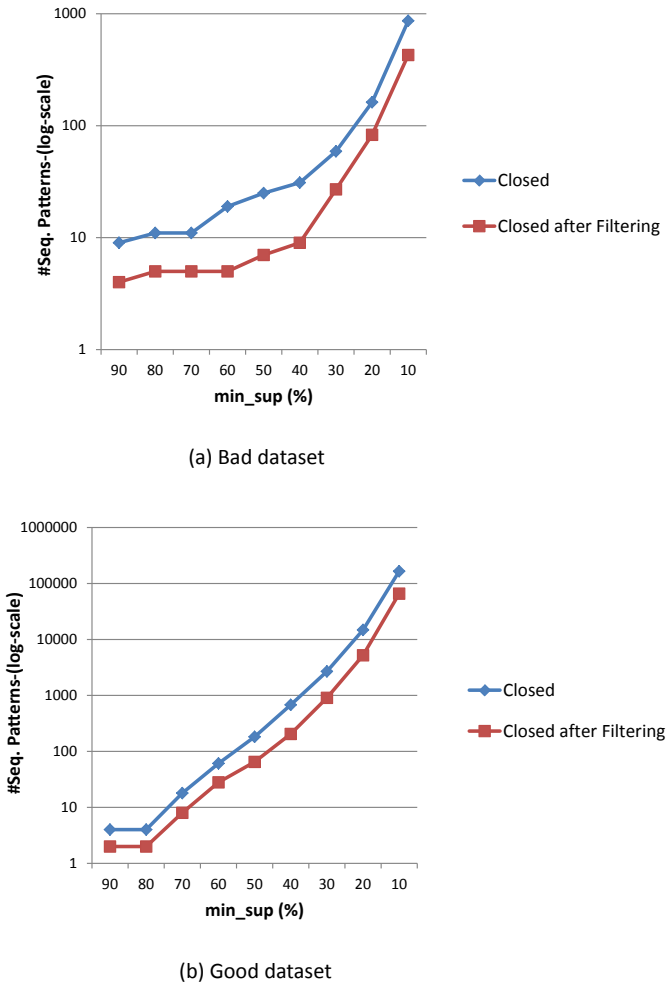


Figure 4.8: Number of the closed sequential patterns in the bad and good datasets before and after filtering.

In Figure 4.9, the number of the anomalies obtained by Equation 4.2 along with the number of the groups that these anomalies are divided into are given. From Figures 4.8 and 4.9, it can be inferred that although the number of the generated closed sequential patterns from the bad and good datasets can be quite high, the number of the anomalies that the user needs to inspect to understand the root cause of the deadlock is mostly less

Datasets	#Traces	#Subtraces	Traces Avg. Len.	Subtraces Avg. Len.	Avg. Len. Red. (%)
bad (Σ_B)	4109	2046	1677	25	98.5%
good (Σ_G)	107029	7258	3079	22	99.3%

Table 4.2: Length reduction results for the POTS model datasets

than 10, at least for thresholds of not less than 20. In Figure 4.9, the precision of the method shows the number of the sequences in the result set which actually reveal some problematic interaction. As this figure shows, only for the thresholds of 30%, 20% and 10% the precision is less than 100%.

Mining closed sequential patterns from the good dataset of POTS with the low value of 10% for min_supp takes 359.651 s and consumes 31.327 MB of main memory while with the low value of 90% for min_supp it takes only 0.074 sec. and consumes only 3.69 MB of main memory.

Considering the way that we generate the good and the bad datasets, these datasets may not include all the possible good and bad traces that can be produced by the execution of the model. In the final result set of the method, therefore, we may get some false positives that do not reveal any problematic behaviors in the model. The computed precision measure for each case study shows the number of the true explanatory sequences among all the sequences of the result set. This precision was calculated manually.

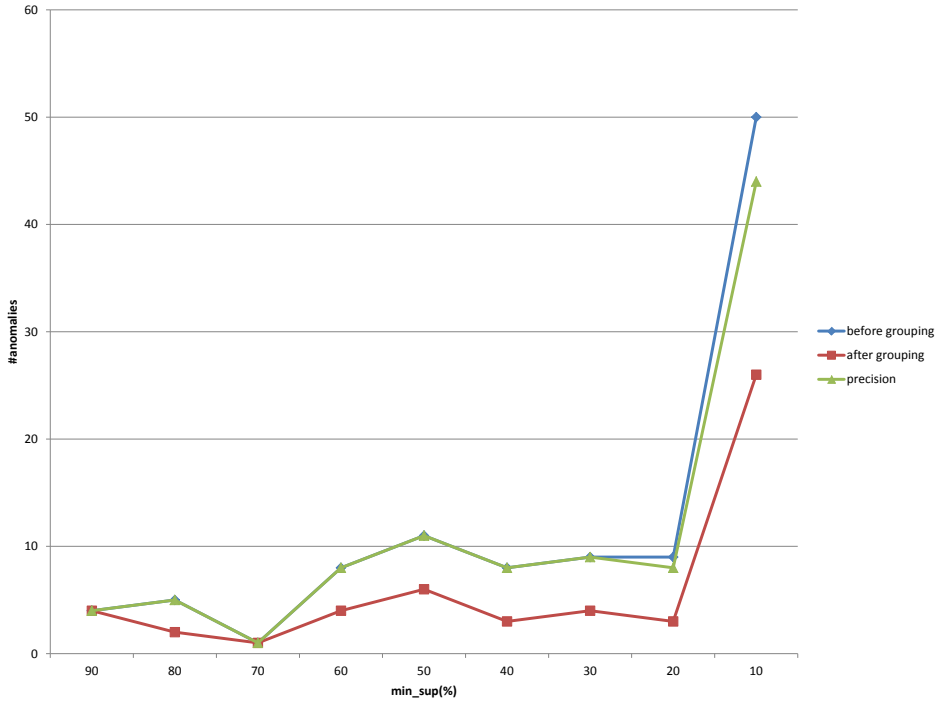


Figure 4.9: Number of the anomalies, number of the groups of anomalies and the precision

The manual inspection of the explanatory sequences in the final result set of the method reveals some faults in the model. In fact, two faults can be detected from the result sets generated by the thresholds 20% and 10%. Other result sets which are generated by higher threshold values only reveal one fault. For example, one of the explanatory sequences computed with min_supp = 90% is given in Figure 4.10.

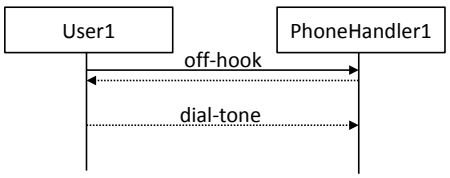


Figure 4.10: An explanatory sequence for POTS

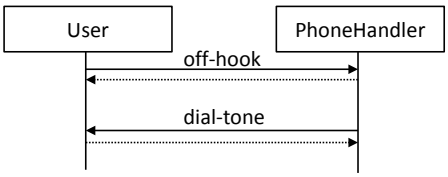


Figure 4.11: Part of a scenario in the POTS model

According to the scenarios defined for this model, the expected sequence from the model designer perspective is shown in Figure 4.11. Considering the expected sequence the receiving of dial-tone message should always be preceded by the sending of dial-tone message. The explanatory sequence in Figure 4.10 reveals a deviation from the expected sequence in Figure 4.11 because in the explanatory sequence the receiving of dial-tone message is not preceded by a corresponding sending of this message. This implicitly reveals the presence of an unread message in the channel from PhoneHandler1 to User1. Finally, it can be inferred that there is a lack of synchronization between the user and the phone handler proctypes so that when the phone handler sends a dial-tone message, the user instead of receiving that message takes another action.

It must be noted that our method is not supposed to be complete, and we use the method as part of an iterative debugging process. After each run of the method, aided by the revealed anomalies the user will try to remove as many causes of property violation as possible. In case the model still contains faults after being modified, the user will apply the method again. This procedure can be iterated until all the causes of property violation in the model have been removed. For example, we tried to remedy the problem in POTS by adding some code in the user proctype which removes a message dial-tone from the channel from the phone handler proctype to the user proctype, if it is present, when sending an on-hook message. After this modification, we again applied our method on the resulting model, this time the number of the generated counterexamples decreased from 4109 to 2229. The produced result set reveals that there is still a lack of synchronization between the user and the phone handler proctypes.

Case Study 2: Rether Model. The second model is a Real-time Ethernet protocol named Rether. It was obtained from [Pel06]. In order to reduce the size and complexity of the original model from [Pel06] we have reduced the values of its parameters. A detailed description of this model can be found in Section 3.1.2 of Chapter 3. We applied our method to this model in order to explain the occurrence of a deadlock. As we explained in Section 4.2.2, the statement i = 0 in Token proctype was used for breaking the traces because all the interactions between the processes in this model starts with the execution of this statement. Table 4.3 shows the amount of the length reduction of the traces for this case study.

Datasets	#Traces	#Subtraces	Traces Avg. Len.	Subtraces Avg. Len.	Avg. Le. Red. (%)
bad (Σ_B)	8	20	322	26	92%
good (Σ_G)	78	51	298	25	92%

Table 4.3: Results of length reduction for the Rether model datasets

The result of mining anomalies with min_supp = 2% is given in Table 4.4. This Table shows the number of the computed closed patterns before and after filtering, the number of the anomalies computed by Equation 4.2, the number of groups of anomalies, the number of the faults detected using the anomalies through the inspection by user, and the precision of the method.

Datasets	#Closed Patterns	#Filtered Closed Patt.	#Anomalies	#Anomaly Groups	Precision	#Detected Faults
bad	182	170	23	11	7	1
good	466	244				

Table 4.4: Mining results for the Rether model

Even though approximately 65% of the anomaly groups in the final result set reveal a problematic behavior in the system, the inspection of only two of them, corresponding to the first and to the 8^{th} groups in the ranked result set, is required for localizing an atomicity violation in one of the proctypes of the model. In Section 3.1.2 of Chapter 3, it is extensively discussed which specific sequence of events reveals an atomicity violation in this model. This problematic sequence of events correspond to two patterns (first and 8^{th}) in the result set computed by min_supp = 2% for this case study.

Comparison with the Method of Chapter 3. The fault localization method that we proposed in Chapter 3 aids the user in locating unforeseen interleavings inside the counterexamples of concurrent systems by extracting a set of short substrings of mainly length two that only occur in the bad dataset. These short substrings along with the corresponding counterexamples are given to the user for further analysis. For example, for this case study, this method generates 3 short distinguishing substrings of length two which are given to the user along with the corresponding counterexamples. Using these substrings, the user needs to inspect on the average 30 events inside the corresponding counterexamples in order to identify the anomalous sequences pointing to an atomicity violation bug in the model. However, the explanatory sequences detected with the method proposed in this chapter are in themselves indicative of the atomicity violation bug in the model. In other words, as opposed to the method of Chapter 3 an inspection of counterexamples is not required. Specifically, in order to detect an atomicity violation in this case study, an explanatory sequence of at least length 30 needs to be isolated inside a counterexample. With the aid of the short substrings of length two extracted by the method of Chapter 3, the user still needs to inspect the counterexample in order to isolate an explanatory sequence of length 30, even though these substrings facilitate

the user inspection greatly. However, the groups of anomalies generated by the method of this paper contain the explanatory sequence of length 30 required for locating the atomicity violation in the model. In fact, the last 7 events of this sequence appear in the first group of the ranked result set and the rest of the events appear in the 8^{th} group. Therefore, the current method imposes less inspection effort on the user for locating the faults in the model.

Case Study 3: Railway Model. We finally applied our method to explain counterexamples indicating the violation of a safety property in the small railroad crossing example which is also used as a sample case study in [LFL13]. The desired safety property is that the car and the train should never be in the crossing simultaneously, which is considered a hazardous state of the system. In this small model, the length reduction step was not necessary.

Table 4.5 summarizes the results of mining obtained with "min_supp = 90%". Columns "#Traces" and "Avg. Trace Len." show the number of traces and the average trace length in the bad and good datasets, respectively.

Datasets	#Traces	Avg. Trace Len.	#Closed Patterns	#Filtered Closed Patt.	#Anomalies	#Detected Faults
bad	28	15	1	1	1	1
good	85	15	6	2		

Table 4.5: Results of mining for the Railway model

The only extracted explanatory sequence reveals a problematic sequence of events that leads the system to an undesired state in which the variables carcrossing and traincrossing have both the value "1". This indicates that both a car and a train are in the crossing at the same time, which is equivalent to a hazard state. This sequence, in fact, guides the user to the location of an atomicity violation bug in the Gate proctype. The presumed intention of the model designer is that the transmission of the signal "1" through the gateCtrl channel would be performed atomically with the changing of the global variable gateStatus to "1". However, due to the fault in the model, the execution of these two statements is interleaved with some other concurrent actions and leads the system to a hazard state.

Dataset Generation. As it has been explained in Section 3.2.1 of Chapter 3, we use the option "-c0 -e" in Spin, for generating the good and the bad datasets which can be time-consuming for some case studies. For example, for the POTS model Spin generates 303,589 good traces which takes around 14 hours. However, the dataset generation for the other two case studies takes less than a minute. If the generated datasets have fewer numbers of traces than the ones generated with the option "-c0 -e", our method is still applicable to them since the method is not guaranteed to be complete. However, when the datasets offer a higher coverage of the good and the bad behaviors, the output of the

method is more precise and the number of the false positives among the explanations is reduced. Moreover, more than one fault can be detected.

4.4 Related Work

In this section, we briefly discuss closely related work that has not yet been addressed in earlier sections.

Pattern Mining in Software Analysis. Data mining techniques have proven to be useful in the analysis of very large amounts of data produced in the course of different activities during various states of the software system development cycle. Frequent pattern mining techniques which find commonly occurring patterns in a dataset are broadly used for mining specifications and localizing faults in program code [LKL07, LYY+05, PNK11, FLS06, RGJ07]. The work documented in [LKL07] adapts sequential pattern mining techniques in order to mine specifications from recorded traces of software system executions. It seems that the patterns generated by this method can also be used for counterexample explanation. However, we faced scalability issues when applying this method to the POTS model case study that we introduce in Section 4.1.1. The longest distinguishing patterns between the bad and the good datasets that could be generated by this method were only 2 events long and did not carry any interesting information with respect to ordering relationships amongst events. Chronicler [RGJ07] is a static analysis tool which infers function precedence protocols defining ordering relationships among function calls in program code. For extracting these protocols a sequence mining algorithm is used. The methods in [LYY+05, PNK11, FLS06] use graph and tree mining algorithms for localizing faults in sequential program code. A commonality of these methods is that they first construct behavior graphs such as function call graphs from execution traces. They then apply a frequent graph or tree mining algorithm on the passing and failing datasets of constructed graphs in order to determine the suspicious portions of the sequential program code. As opposed to this approach, our goal is to identify sequences of interleaved actions in concurrent systems, which the above cited works are unable to provide.

Concurrency Bug Detection Methods. Avio [LTQZ06] only detects atomicity violations and, as opposed to our method, is tailored to only identify single variable bugs. Examples of tools which only focus on detecting data races are lockset bug detection tools [SBN+97b] and happens-before bug detection tools [NM91a]. In contrast to these approaches, which lack generality and rely on heuristics that are specific to a class of bugs, the output of our method in the form of explanatory sequences can be indicative to any type of concurrency bugs in the program design that can be characterized by a reachability property.

The work described in [LC09] proposes a more general approach for finding concurrency bugs based on constructing context-aware communication graphs from execution traces.

Context-aware communication graphs use communication context to encode access ordering information. A key challenge of this method is, however, that if the relevant ordering information is not encoded, bugs may not lead to graph anomalies and therefore remain undetected. Our method does not rely on such an encoding but directly analyzes the temporal ordering of the event. It therefore appears to be more general than the approach in [LC09].

For a more detailed discussion of concurrency bug detection tools and techniques such as the ones mentioned in this section see Chapter 2, Section 2.2.1.

Counterexample Explanation Methods. In Section 3.4 of Chapter 3, we provide a detailed comparison of our method with a closely related work by Groce and Visser [GV03]. For that comparison, the arguments given in Section 3.4 of Chapter 3 are also valid for the work of this chapter, because, the current method is the enhancement of our precursory work. The *causality checking* method proposed in [LFL13] computes automatically the causalities in system models by adapting the counterfactual reasoning based on the *structural equation model (SEM)* by Halpern and Perl [HP05]. This method identifies sequences of events that cause a system to reach a certain undesired state by extending depth-first search and breadth-first search algorithms used for a complete state space exploration in explicit-state model checking. It seems that the main superiority of our method is less computational cost in terms of memory and running time for detecting at least one fault in the model. As we have shown in Section 4.3, our method very efficiently detected a fault in the sample case study (Railway model) of [LFL13]. The *causality checking* method considers all the possible finite good and bad execution traces for identifying the combination of events which are causal for the violation of a safety property. Since we do not seek completeness, our mining method is still applicable even if the datasets do not include all the possible good and bad execution traces, which can be an impediment in practice.

Some other automated counterexample explanation techniques such as [BBDC+09, WYIG06, GCKS06] only take the values of program or model variables into account when computing which variable values along a counterexample trace cause a violation of some desired property. In contrast, the method we propose here considers the order of execution of actions and can hence explain property violations which are due to a specific order of execution of actions.

4.5 Summary

In this chapter, we have presented an anomaly detection method for the explanation of model checking counterexamples for message passing concurrent system models. From a dataset of counterexamples we extract a number of explanatory sequences of events that prove to point to the location of the fault in the model by leveraging a frequent pattern mining technique called sequential pattern mining. An experimental analysis showed the effectiveness of our method for a number of indicative deadlock checking case studies.

CHAPTER 5

Abstraction and Mining of Traces to Explain Concurrency Bugs

In Chapters 3 and 4, we presented two anomaly detection based techniques for explaining counterexamples of *message passing* concurrent systems. In these techniques, anomalies in the form of sequences of events which referred to as *explanatory sequences* are extracted using a mining method called *sequential pattern mining*. The computed anomalies reveal the problematic interleavings in concurrent systems which result in property violation. In this chapter, we propose an anomaly detection technique for explaining concurrency bugs in shared memory concurrent programs. In this technique, we analyze the execution traces of multithreaded concurrent programs in order to isolate the problematic order between the execution of different threads which is causal for the failure. As we have seen in Chapter 2, *data races*, *atomicity violations* and *order violations* are the common types of non-deadlock concurrency bugs that occur in shared memory concurrent programs.

Similar to the techniques proposed in Chapters 3 and 4, the technique of this chapter for explaining concurrency bugs is oblivious to the nature of the specific bug. We assume that we are given a set of concurrent execution traces, each of which is classified as successful or failed. This is a reasonable assumption if the program is systematically tested and the test suite satisfies concurrent coverage metrics [LJZ07]. Execution traces can be generated and recorded using systematic testing tools [MQ06, MQB07, YCGK07] or stress testing [PLZ09]. Inspecting concurrent traces manually, however, is still tedious and time-consuming. An empirical study of strategies commonly used for diagnosing and correcting faults in concurrent software shows that the primary concern of the programmer is to produce and analyze a failing trace by reasoning about potential thread interleavings based on some degree of program understanding [FKS+08]. In light of the complexity of this task, tool support is highly desirable.

Although the traces of concurrent programs are lengthy sequences of events, only a small

subset of these events is typically sufficient to explain an erroneous behavior. In general, these events do not occur consecutively in the execution trace, but rather at an arbitrary distance from each other as we have discussed in the previous two chapters. Therefore, we use data mining algorithms to isolate ordered sequences of non-contiguous events which occur frequently in the traces. Subsequently, we isolate anomalies by examining the *differences* between the common behavioral patterns of failing and passing traces (motivated by Lewis' theory of causality and counterfactual reasoning [Lew01]).

Our approach which is based on anomaly detection combines ideas from the fields of runtime monitoring [DGR04], abstraction and refinement [CGJ+00], and sequential pattern mining [ME10]. It comprises the following three phases:

- We systematically generate execution traces with different interleavings, and record all global operations but not thread-local operations [YCGK07], thus requiring only limited observability. We justify our decision to consider only shared accesses in Section 5.1. The resulting data is partitioned into successful and failed executions.
- Since the resulting traces may contain thousands of operations and events, we present a novel abstraction technique which reduces the length of the traces as well as the number of events by mapping sequences of concrete events to single abstract events. We show in Section 5.2 that this abstraction step preserves all original behaviors while reducing the number of patterns to consider significantly.
- We use a sequential pattern mining algorithm [YHA03, WH04] to identify sequences of events that frequently occur in failing execution traces. In a subsequent filtering step, we eliminate from the resulting sequences spurious patterns that are an artifact of the abstraction and misleading patterns that do not reflect problematic behaviors. The remaining patterns are then ranked according to their frequency in the passing traces, where patterns occurring in failing traces exclusively are ranked highest.

In Section 5.4, we use a number of case studies to demonstrate that our approach yields a small number of relevant patterns which can serve as an explanation of the erroneous program behavior.

The work in this chapter improves and extends our previous work [TBWW14] in the following ways:

- We formalize the notion of a *bug explanation pattern*.
- In Section 5.3, we lift the notion of bug explanation patterns to the patterns mined from abstract traces.
- The algorithm for producing bug explanation patterns is presented in Section 5.3.1, followed by a discussion of the parameters of the method and their effects. This section also describes an optimization of the computationally costly filtering step of [TBWW14], resulting in orders of magnitude speed up in run time.

- In the section on experimental results, we demonstrate that our modification of the method in [TBWW14] preserves the effectiveness of the method while achieving more efficiency. Moreover, we show the effect of variations in the input datasets of traces on the effectiveness of the method by bounding the number of context switches in input traces.

5.1 Executions, Failures, and Bug Explanation Patterns

In this section, we define basic notions such as executions, events, traces, and faults. We introduce the notion of bug explanation patterns and provide a theoretical rationale as well as an example of their usage. We recap the terminology of sequential pattern mining and explain how we apply this technique to extract bug explanation patterns from sets of traces.

5.1.1 Programs and Failing Executions

We consider shared-memory concurrent programs composed of k *threads* with indices $\{1, \ldots, k\}$ and a finite set $\mathbb{G}$ of *shared variables*. Each thread T_i where $1 \leq i \leq k$ has a finite set of *local variables* $\mathbb{L}_i$. The set of all variables is then defined by $\mathbb{V} \stackrel{\text{def}}{=} \mathbb{G} \cup \bigcup_i \mathbb{L}_i$, where $1 \leq i \leq k$. Interaction between the threads happens via read and write accesses to shared variables. Each thread is represented by a control flow graph whose edges are annotated with atomic instructions. We use guarded statements to represent atomic instructions. Let $\mathbb{V}_i = \mathbb{G} \cup \mathbb{L}_i$ (for $1 \leq i \leq k$) denote the set of variables accessible in thread T_i. An instruction from thread T_i is either a guarded statement $\mathsf{assume}(\varphi) \triangleright \tau$ or an assertion $\mathsf{assert}(\varphi)$ where φ is a predicate over $\mathbb{V}_i$ and τ is an assignment of the form $v := \phi$ (where $v \in \mathbb{V}_i$ and ϕ is an expression over $\mathbb{V}_i$). The condition φ must be true for the assignment τ to be executed. It must be also true when $\mathsf{assert}(\varphi)$ is executed, otherwise a failure occurs.

The guarded statement has the following three variants: (1) when the guard $\varphi = \mathsf{true}$, it can model ordinary assignments in a basic block, (2) when the assignment τ is empty, the conditions $\mathsf{assume}(\varphi)$ and $\mathsf{assume}(\neg\varphi)$ can model the execution of a branching statement $\mathsf{if}(\varphi) - \mathsf{else}$, and (3) with both the guard and the assignment, it can model an atomic *check-and-set* operation, which is the foundation of all types of concurrency primitives [HS08]. For example, acquiring and releasing a lock l in a thread with index i is modeled as $\mathsf{assume}(l = 0) \triangleright l := i$ and $\mathsf{assume}(l = i) \triangleright l := 0$, respectively. Fork and join can be modeled in a similar manner using auxiliary synchronization variables.

Each thread executes a sequence of atomic instructions in *program order* (determined by the control flow graph). During the execution, the scheduler picks a thread and executes the next atomic instruction in the program order of the thread. The execution halts if there are no more executable atomic instructions.

Executions. An *execution* $\rho = S_0, a_1, S_1, ..., S_{n-1}, a_n, S_n$ is an alternating sequence of states S_i and atomic execution steps a_i corresponding to some interleaving of instructions from the threads of the program. Each state S is a valuation of the variables $\mathbb{V}$. Execution steps correspond to the execution of atomic instructions of the threads. For each i, the execution of a_i in state S_{i-1} leads to state S_i.

The sequence of states visited during an execution constitutes a program behavior. As we have seen in Chapter 1, Section 1.2.1, a *fault* or *bug* is a defect in the program code, which if triggered leads to an *error*, which in turn is a discrepancy between the actual and the intended behavior (specified by assertions or test cases). If an error propagates, it may eventually lead to a *failure*, a behavior contradicting the specification. We call executions leading to a failure *failing* and all other executions *passing* executions.

5.1.2 Read-Write Events and Traces

Each execution of an atomic instruction $\mathsf{assume}(\varphi) \triangleright v := \phi$ in a thread such as T_i generates read events for the variables referenced in φ and ϕ, followed by a write event for v.

Definition 11 (Read-Write Events)**.** *A* read-write event *is a tuple* $\langle \mathsf{id}, \mathsf{tid}, \ell, \mathsf{type}, \mathsf{addr} \rangle$*, where* id *is an identifier,* $\mathsf{tid} \in \{1, \ldots, k\}$ *and* ℓ *are the thread identifier and the source code line number of the corresponding instruction,* $\mathsf{type} \in \{R, W\}$ *is the type (or direction) of the memory access, and* $\mathsf{addr} \in \mathbb{V}_{\mathsf{tid}}$ *is the variable accessed.*

Two events have the same identifier id if they are issued by the same thread and agree on the line number of source code, the type, and the address. In the following, for comparing two events we use their ids. Two events e_i and e_j are equal denoted by $e_i = e_j$ if both have the same ids. However, each event in the execution is unique. Therefore, two events with the same id are distinguished by their index in the sequence of an execution. We use $\mathsf{R}_{\mathsf{tid}}(\mathsf{addr}) - \ell$ and $\mathsf{W}_{\mathsf{tid}}(\mathsf{addr}) - \ell$ to refer to read and write events to the object with address addr issued by thread tid at line number ℓ of the source code, respectively.

Two events conflict if they are issued by different threads, access the same shared variable $v \in \mathbb{G}$, and at least one of them is a write to v. Given two conflicting events e_1 and e_2 from two different threads such that e_1 is issued before e_2, we distinguish three cases of inter-thread data-dependency: (a) flow-dependence: e_2 reads a value written by e_1, (b) anti-dependence: e_1 reads a value before it is overwritten by e_2, and (c) output-dependence: e_1 and e_2 both write the same memory location. Figures 5.1 and 5.2 show all inter-thread data-dependencies for the shared variable *balance* in the passing and failing traces of the running example given in Section 5.1.3. We use dep to denote the set of data-dependencies between the events of an execution that arise from the order in which the instructions are executed.

A failing and a passing execution started in the same initial state either (a) differ in their data-dependencies dep over the shared variables, and/or (b) contain different local computations. Local computations of thread T_i involve thread local variables, $v \in \mathbb{L}_i$. In our setting, we assume local computations of the threads of the program are not the

cause of the error. Therefore, in a failing and a passing execution started in the same initial state, a discrepancy in either their data-dependencies dep over the shared variables or the executed events explains the failure in the failing trace according to fundamental results of concurrency control originally developed in database research [Pap79] and Mazurkiewicz's trace theory [Maz86]. This discrepancy is, in fact, induced by the order of execution of the instructions of the program, which is the result of a change in the schedule. (As an example, compare the passing and failing traces given in Figures 5.1 and 5.2.)

Our method aims at identifying sequences of events that reveal this discrepancy. Therefore, we focus on concurrency bugs that manifest themselves in a deviation of the accesses to and the data-dependencies between *shared* variables, thus ignoring failures caused purely by a difference of the local computations. As per the argument above, this criterion covers a large class of concurrency bugs, including data races, atomicity violations, and order violations.

To this end, we log the order of read and write events (for shared variables) in a number of passing and failing executions. Since we are interested in concurrency bugs which are due to scheduling rather than input values, failing and passing traces all start from the same initial state. Moreover, in the logged read/write events we ignore the value of the shared variables. We assume that the addresses of variables are consistent across executions, which is enforced by our logging tool. A trace is then defined as follows:

Definition 12. *A* trace $\sigma = \langle e_1, e_2, ..., e_n \rangle$ *is a finite sequence of read-write events of shared variables (Definition 11).*

In the following, Σ_F and Σ_P denote sets of failing and passing traces, respectively.

5.1.3 Bug Explanation Patterns

In a failing trace, we refer to a sequence of events relevant to the failure as *bug explanation sequence*. We typically can distinguish two types of events in a bug explanation sequence: the events triggering the error (which is a discrepancy between the intended and the actual behavior) and the events propagating the error, eventually leading to a failure. We illustrate these notions (bug explanation sequences, triggering and propagating events) using a well-understood example of an atomicity violation. Figure 5.1 shows two code fragments that non-atomically update the balance of a bank account (stored in the shared variable balance) by depositing or withdrawing given values. The example does not contain a data race, since balance is protected by the lock balance_lock. The global array t_array contains the sequence of amounts to be transferred. Two threads execute these code fragments concurrently. In Figures 5.1 and 5.2, two failing traces and one passing trace resulting from the concurrent execution of the code fragments by two threads are given. The identifiers on (where n is a number) represent the addresses of the accessed shared objects, and o27 corresponds to the variable balance. The events $\mathsf{R}_1(\mathsf{o27}) - 67$ and $\mathsf{W}_1(\mathsf{o27}) - 74$ correspond to the read and write instructions at lines 67 and 74 of thread 1,

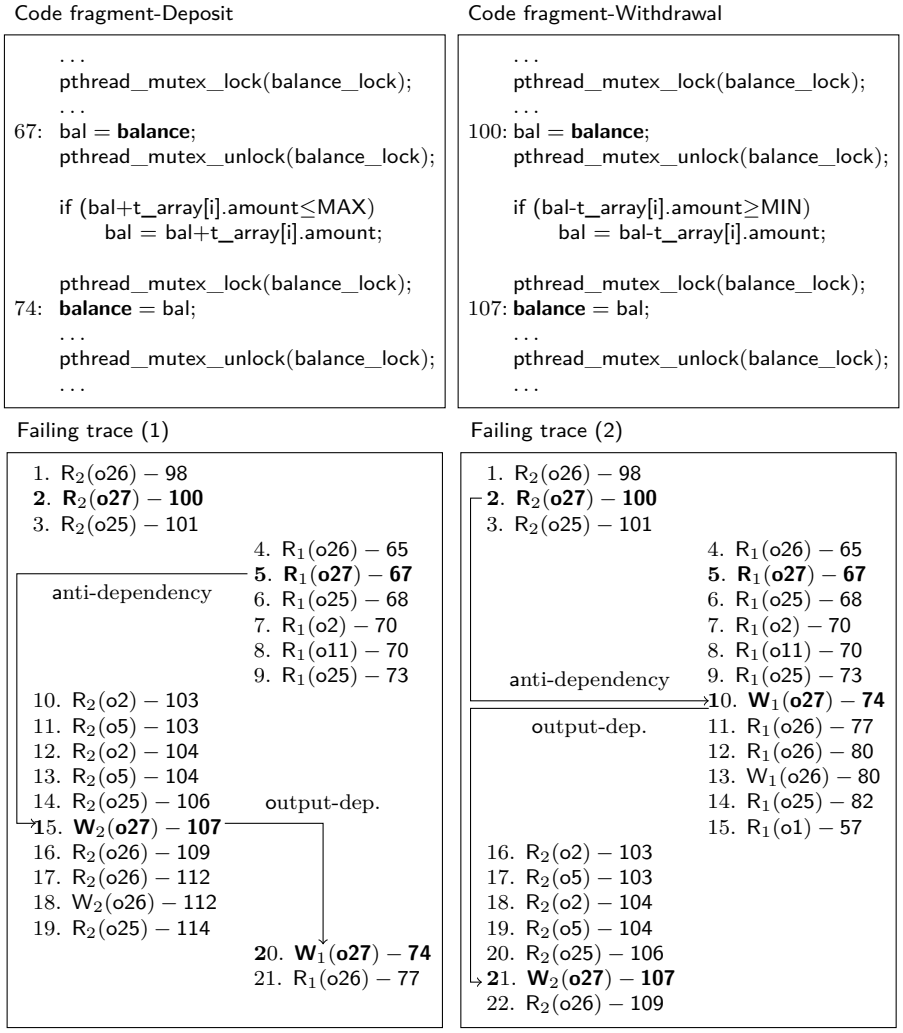


Figure 5.1: Conflicting update of bank account balance

respectively. Similarly, the events $\mathsf{R}_2(\mathsf{o27}) - 100$ and $\mathsf{W}_2(\mathsf{o27}) - 107$ correspond to the read and write instructions at lines 100 and 107 of thread 2, respectively.

The traces in Figure 5.1 fail because their final states are inconsistent with the expected value of balance. For example, in failing trace (1), the reason is that o27 is overwritten with a stale value at position 20 in the trace, "killing" the transaction of thread 2 that writes o27 at position 15. This is reflected by the sequence $\langle \mathsf{R}_1(\mathsf{o27}) - 67, \mathsf{W}_2(\mathsf{o27}) - 107, \mathsf{W}_1(\mathsf{o27}) - 74 \rangle$ in combination with the data-dependencies between the events as depicted in the figure. This sequence reveals the cause of failure and is an example of a bug explanation sequence in which the first two events $\langle \mathsf{R}_1(\mathsf{o27}) - 67, \mathsf{W}_2(\mathsf{o27}) - 107 \rangle$ trigger the error.

Since a single fault can have different manifestations at run time, bug explanation sequences may vary in different failing traces. For example, in Figure 5.1 the failing trace

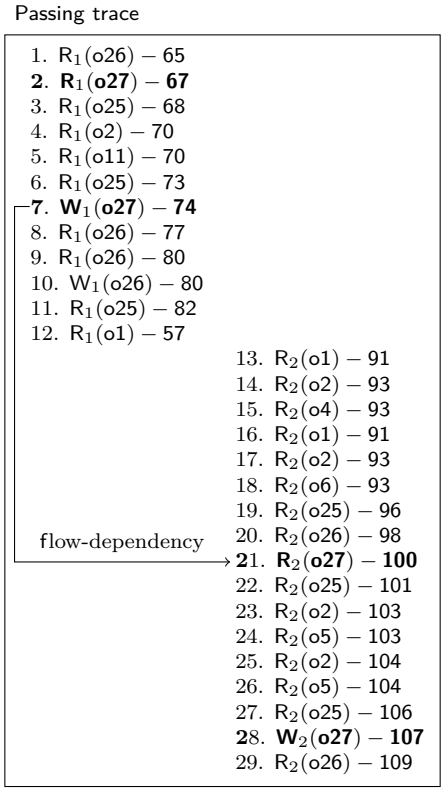


Figure 5.2: Passing trace of the bank account example

(2) which fails due to the same fault as trace (1) has a different bug explanation sequence and consequently different triggering events: $\langle \mathsf{R}_2(\mathsf{o27}) - 100, \mathsf{W}_1(\mathsf{o27}) - 74, \mathsf{W}_2(\mathsf{o27}) - 107 \rangle$ (the first two events trigger the error). The two bug explanation sequences discussed above and the corresponding dependencies do not arise in any passing trace, since no context switch occurs between the events $\mathsf{R}_1(\mathsf{o27}) - 67$ and $\mathsf{W}_1(\mathsf{o27}) - 74$.

Although bug explanation sequences vary in different failing traces (failing traces 1 and 2 in Figure 5.1), in the set Σ_F of failing traces which all fail due to the same fault, bug explanation sequences typically share triggering or propagating events. Assume the code fragments of Figure 5.1 are executed in a loop by the two threads. Some traces in Σ_F will then share $\langle \mathsf{R}_1(\mathsf{o27}) - 67, \mathsf{W}_2(\mathsf{o27}) - 107 \rangle$ as the triggering events, while in some other traces the occurrence of sequence $\langle \mathsf{R}_2(\mathsf{o27}) - 100, \mathsf{W}_1(\mathsf{o27}) - 74 \rangle$ triggers the error.

We refer to the portions of bug explanation sequences that occur commonly in Σ_F as *bug explanation patterns* such as $\langle \mathsf{R}_1(\mathsf{o27}) - 67, \mathsf{W}_2(\mathsf{o27}) - 107 \rangle$ in the running example. Intuitively, these patterns occur more frequently in the failing dataset Σ_F than in the set Σ_P of passing traces. While the bug pattern in question may occur in passing executions (since an error may not necessarily lead to a failure), our approach is based on the assumption that it is less frequent in Σ_P. Therefore, for explaining concurrency bugs we examine the differences in terms of the sequence of events in the traces of the

failing and passing datasets, which is the foundation of a large number of approaches for locating faults in program code (see, for instance, [Zel09]). Lewis' theory of causality and counterfactual reasoning provides justification for this type of fault localization approaches [Lew01].

Since our focus is on concurrency bugs which are due to problematic interactions between threads, the triggering events are from at least two different threads and do not necessarily occur consecutively inside the trace. In general, these events can occur at an arbitrary distance from each other due to scheduling. Our bug explanation patterns are therefore, in general, *subsequences* of execution traces. According to Definition 5, $\pi = \langle e'_0, e'_1, e'_2, ..., e'_m \rangle$ is a *subsequence* of $\sigma = \langle e_0, e_1, e_2, ..., e_n \rangle$, denoted as $\pi \sqsubseteq \sigma$, if and only if there exist integers $0 \leq i_0 < i_1 < i_2 < i_3 ... < i_m \leq n$ such that $e'_0 = e_{i_0}, e'_1 = e_{i_1}, ..., e'_m = e_{i_m}$. We write $\pi \sqsubset \sigma$ if $\pi \sqsubseteq \sigma$ and $\pi \neq \sigma$. We also call σ a *super-sequence* of π if $\pi \sqsubseteq \sigma$.

5.1.4 Mining Bug Explanation Patterns

In order to isolate bug explanation patterns in the traces of Σ_F, we use *sequential pattern mining* algorithms. As we have already seen in Chapters 3 and 4, these algorithms extract frequent subsequences from a dataset of sequences without limitations on the relative distance of events belonging to the subsequences. This data mining technique has diverse applications in areas such as the analysis of customer purchase behavior, the mining of web access patterns or motifs in DNA sequences.

In Chapters 3 and 4, we have introduced the terminology of sequential pattern mining. In this section, we recap it again in order to make the presentation self-contained and adapt it to our setting. For a more detailed treatment, we refer the interested reader to [ME10]. In our setting, we are interested in extracting subsequences occurring frequently in Σ_F and contrasting them with the frequent subsequences of Σ_P. As we have already discussed, bug explanation patterns are subsequences which occur more frequently in the failing dataset Σ_F.

In a sequence dataset $\Sigma = \{\sigma_1, \sigma_2, ..., \sigma_n\}$, a pattern is supported by a sequence if it is a subsequence of it. The *support* of a sequence π is defined as

$$\mathsf{support}_\Sigma(\pi) \stackrel{\text{def}}{=} |\{\sigma \,|\, \sigma \in \Sigma \wedge \pi \sqsubseteq \sigma\}| \ .$$

Given a minimum support threshold $\mathsf{min\_supp}$, the pattern π is considered a sequential pattern or a frequent subsequence if $\mathsf{support}_\Sigma(\pi) \geq \mathsf{min\_supp}$. $\mathrm{FS}_{\Sigma,\mathsf{min\_supp}}$ denotes the set of all sequential patterns mined from Σ with the given support threshold $\mathsf{min\_supp}$ and is defined as $\mathrm{FS}_{\Sigma,\mathsf{min\_supp}} = \{\pi \,|\, \mathsf{support}_\Sigma(\pi) \geq \mathsf{min\_supp}\}$. As an example, Σ contains the four traces given in Table 5.1. We obtain:

Id	Trace
1	$\mathsf{R}_1(\mathsf{x})$,$\mathsf{W}_1(\mathsf{x})$,$\mathsf{R}_2(\mathsf{x})$,$\mathsf{W}_2(\mathsf{x})$,$\mathsf{R}_1(\mathsf{x})$,$\mathsf{W}_1(\mathsf{x})$
2	$\mathsf{R}_1(\mathsf{x})$,$\mathsf{W}_1(\mathsf{x})$,$\mathsf{R}_1(\mathsf{x})$,$\mathsf{W}_1(\mathsf{x})$,$\mathsf{R}_2(\mathsf{x})$,$\mathsf{W}_2(\mathsf{x})$
3	$\mathsf{R}_1(\mathsf{x})$,$\mathsf{R}_2(\mathsf{x})$,$\mathsf{W}_1(\mathsf{x})$,$\mathsf{W}_2(\mathsf{x})$,$\mathsf{R}_1(\mathsf{x})$,$\mathsf{W}_1(\mathsf{x})$
4	$\mathsf{R}_2(\mathsf{x})$,$\mathsf{R}_1(\mathsf{x})$,$\mathsf{W}_2(\mathsf{x})$,$\mathsf{W}_1(\mathsf{x})$,$\mathsf{R}_1(\mathsf{x})$,$\mathsf{W}_1(\mathsf{x})$

Table 5.1: Sample dataset of traces

$$
\begin{aligned}
\mathrm{FS}_{\Sigma,4} \;=\; \{&\langle \mathsf{R}_1(\mathsf{x})\rangle : 4,\\
&\langle \mathsf{R}_2(\mathsf{x})\rangle : 4,\\
&\langle \mathsf{W}_1(\mathsf{x})\rangle : 4,\\
&\langle \mathsf{W}_2(\mathsf{x})\rangle : 4,\\
&\langle \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x})\rangle : 4,\\
&\langle \mathsf{R}_1(\mathsf{x}), \mathsf{W}_2(\mathsf{x})\rangle : 4,\\
&\langle \mathsf{R}_2(\mathsf{x}), \mathsf{W}_2(\mathsf{x})\rangle : 4,\\
&\langle \mathsf{W}_1(\mathsf{x}), \mathsf{R}_1(\mathsf{x})\rangle : 4,\\
&\langle \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x}), \mathsf{R}_1(\mathsf{x})\rangle : 4,\\
&\langle \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x})\rangle : 4,\\
&\langle \mathsf{R}_1(\mathsf{x}), \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x})\rangle : 4,\\
&\langle \mathsf{W}_1(\mathsf{x}), \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x})\rangle : 4,\\
&\langle \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x}), \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x})\rangle : 4\}
\end{aligned}
$$

where the numbers following the patterns denote the respective supports of the patterns.

Notice the combinatorial number of the frequent subsequences even in this small dataset. In order to avoid a combinatorial explosion, it is best to mine *closed* set of patterns [YHA03, WH04]. In $\mathrm{FS}_{\Sigma,4}$, patterns $\langle \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x}), \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x})\rangle : 4$ and $\langle \mathsf{R}_2(\mathsf{x}), \mathsf{W}_2(\mathsf{x})\rangle : 4$, which do not have any super-sequences with the same support value are called *closed* patterns. A closed pattern encompasses all the frequent patterns with the same support value which are all subsequences of it. For example, in $\mathrm{FS}_{\Sigma,4}$, $\langle \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x}), \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x})\rangle : 4$ encompasses $\langle \mathsf{R}_1(\mathsf{x})\rangle : 4$, $\langle \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x})\rangle : 4$, $\langle \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x}), \mathsf{R}_1(\mathsf{x})\rangle : 4$ and similarly $\langle \mathsf{R}_2(\mathsf{x}), \mathsf{W}_2(\mathsf{x})\rangle : 4$ encompasses $\langle \mathsf{R}_2(\mathsf{x})\rangle : 4$ and $\langle \mathsf{W}_2(\mathsf{x})\rangle : 4$. Closed patterns are the lossless compression of all sequential patterns. Therefore, in our method we mine only *closed* patterns in order to avoid a combinatorial explosion. $\mathrm{CS}_{\Sigma,\mathsf{min\_supp}}$ denotes the set of all closed sequential patterns mined from Σ with the support threshold $\mathsf{min\_supp}$ and is defined as

$$\{\pi \mid \pi \in \mathrm{FS}_{\Sigma,\mathsf{min\_supp}} \wedge \nexists \pi' \in \mathrm{FS}_{\Sigma,\mathsf{min\_supp}} \,.\, \pi \sqsubset \pi' \wedge \mathsf{support}(\pi) = \mathsf{support}(\pi')\}.$$

To extract bug explanation patterns from Σ_P and Σ_F, we first mine closed sequential patterns with a given minimum support threshold $\mathsf{min\_supp}$ from Σ_F. At this point, we ignore the index of events in execution traces and identify events using their id. This is because in mining we do not distinguish between events with the same id that occur at different positions inside a trace. The event $\mathsf{W}_1(\mathsf{o27}) - \mathsf{74}$ in Figure 5.1, for instance, has the same id in the failing traces and the passing trace, even though its indices in these traces (20, 10 and 7) differ.

To determine whether a pattern π in $\mathrm{CS}_{\Sigma_F,\mathsf{min\_supp}}$ is more frequent in Σ_F than in Σ_P, we define the notion of *relative support* which is computed as the following:

$$\mathsf{rel\_supp}(\pi) = \frac{\mathsf{support}_{\Sigma_F}(\pi)}{\mathsf{support}_{\Sigma_F}(\pi) + \mathsf{support}_{\Sigma_P}(\pi)}.$$

Note that the values of support in Σ_F and Σ_P are normalized. Patterns that occur in Σ_F exclusively have the maximum relative support of 1. Patterns that occur with the same frequency in both Σ_F and Σ_P have the relative support of 0.5. Therefore, from $\mathsf{rel\_supp}(\pi) > 0.5$ we infer that π occurs more frequently in Σ_F than in Σ_P. We argue that the patterns with the highest relative support are indicative of one or several faults inside the program of interest. These patterns can hence be used as clues for the exact location of the faults inside the program code.

Sequential pattern mining ignores the underlying semantics of the events. This has the undesirable consequences that we obtain numerous patterns that are not explanations in the sense of Section 5.1.3, since they do not contain context switches or data-dependencies. In $\mathrm{FS}_{\Sigma,4}$, $\langle \mathsf{R}_2(\mathsf{x}), \mathsf{W}_2(\mathsf{x})\rangle : 4$ does not contain any context switches, hence cannot be a candidate bug explanation pattern. Pattern $\langle \mathsf{R}_1(\mathsf{x}), \mathsf{W}_2(\mathsf{x})\rangle : 4$ occurs in all four traces of Σ, however only in trace 4 the two events are anti-dependent. In all other traces, they are not related by any data-dependencies. Accordingly, we define heuristics to consider a pattern as a *candidate bug explanation pattern.*

Definition 13 (Bug Explanation Pattern)**.** *Given Σ_F and Σ_P and* $\mathsf{min\_supp}$*, pattern $\pi \in CS_{\Sigma_F,\mathsf{min\_supp}}$ is a* candidate bug explanation pattern *if* $\mathsf{rel\_supp}(\pi) > 0.5$ *and $\forall e_i \in \pi, \exists e_j \in \pi, i \neq j$ such that e_i and e_j are related by* dep*. In addition, at least two related events should belong to two different threads.*

In our method, the heuristics defined in Definition 13 are applied to the patterns of $\mathrm{CS}_{\Sigma_F,\mathsf{min\_supp}}$ in a post-processing step after mining. This process involves mapping of $\pi \in \mathrm{CS}_{\Sigma_F,\mathsf{min\_supp}}$ to the traces in Σ_F for locating the *instances* of π in these traces. At this point, the index of events inside the traces is taken into account (indices $\ell_1, \ell_2, \ldots, \ell_m$ in Definition 14).

Definition 14 (Instance of a Pattern in a Trace)**.** *$I(\ell_1, \ell_2, \ldots, \ell_m)$ is an* instance *of pattern $\pi = \langle e'_1, e'_2, ..., e'_m\rangle$ in the trace $\sigma = \langle e_1, e_2, ..., e_n\rangle$ if $e'_1 = e_{\ell_1}, e'_2 = e_{\ell_2}, \ldots, e'_m = e_{\ell_m}$ where $1 \leq \ell_i \leq n$ for $1 \leq i \leq m$.*

Support Thresholds and Datasets. Which threshold is adequate depends on the number and the nature of the bugs. Given a single fault involving only one variable, most traces in Σ_F presumably share the same sequence of events that trigger the error. Since the bugs are not known up-front, and lower thresholds result in a larger number of patterns, we gradually decrease the threshold until bug explanations emerge. Moreover, the quality of the explanations is better if the traces in Σ_P and Σ_F are similar or homogeneous in terms of events they contain and the order between them. Our experiments in Section 5.4

show that the sets of execution traces need not necessarily be exhaustive to enable bug explanations.

5.2 Mining Abstract Execution Traces

With increasing length of the execution traces and number of events, sequential pattern mining quickly becomes intractable [LTB12] (see Chapter 3, Section 3.1.4). To alleviate this problem, we introduce *macro-events* that represent events of the same thread occurring consecutively inside an execution trace, and obtain *abstract* events by grouping these macros into equivalence classes according to the events they replace. Our abstraction reduces the length of the traces as well as the number of the events at the cost of introducing spurious traces. Accordingly, patterns mined from the abstract traces may not occur as a subsequence of any original traces. Therefore, we eliminate spurious patterns using a subsequent feasibility check.

5.2.1 Abstracting Execution Traces

In order to obtain a more compact representation of a set Σ of execution traces, we introduce *macros* representing substrings of the traces in Σ. A substring of a trace σ is a sequence of events that occur consecutively in σ.

Definition 15 (Macros)**.** *Let Σ be a set of execution traces. A* macro-event *(or* macro*, for short) is a sequence of events $m \stackrel{\text{def}}{=} \langle e_1, e_2, ..., e_k \rangle$ in which all the events e_i $(1 \leq i \leq k)$ have the same thread identifier, and there exists $\sigma \in \Sigma$ such that m is a substring of σ.*

We use $\mathsf{events}(m)$ to denote the set of events in a macro m. The concatenation of two macros $m_1 = \langle e_i, e_{i+1}, \ldots e_{i+k} \rangle$ and $m_2 = \langle e_j, e_{j+1}, \ldots e_{j+l} \rangle$ is defined as $m_1 \cdot m_2 = \langle e_i, e_{i+1}, \ldots e_{i+k}, e_j, e_{j+1}, \ldots e_{j+l} \rangle$. We denote the concatenation of a sequence of macros $\Pi = \langle m_1, m_2, \ldots m_l \rangle$ as $\mathsf{concat}(\Pi) = m_1 \cdot m_2 \cdots m_l$.

Definition 16 (Macro trace)**.** *Let Σ be a set of execution traces, $\mathbb{E}$ the set of events occurred in traces of Σ, and $\mathbb{M}$ be a set of macros. Given $\sigma \in \Sigma$, a corresponding* macro trace *$\langle m_1, m_2, \ldots, m_n \rangle$ is a sequence of macros $m_i \in \mathbb{M}$ $(1 \leq i \leq n)$ such that $m_1 \cdot m_2 \cdots m_n = \sigma$. We say that $\mathbb{M}$* covers *Σ if there exists a corresponding macro trace (denoted by $\mathsf{macro}(\sigma)$) for each $\sigma \in \Sigma$. Moreover, we use $\mathsf{macro}(\Sigma)$ to denote a set of macro traces corresponding to Σ.*

Note that the mapping $\mathsf{macro} : \mathbb{E}^+ \to \mathbb{M}^+$ is not necessarily unique. Given a mapping macro, every macro trace can be mapped to an execution trace and vice versa. For example, for $\mathbb{M} = \{m_0 \stackrel{\text{def}}{=} \langle e_0, e_2 \rangle, m_1 \stackrel{\text{def}}{=} \langle e_1, e_2 \rangle, m_2 \stackrel{\text{def}}{=} \langle e_3 \rangle, m_3 \stackrel{\text{def}}{=} \langle e_4, e_5, e_6 \rangle, m_4 \stackrel{\text{def}}{=}$

$\langle e_8, e_9\rangle, m_5 \stackrel{\text{def}}{=} \langle e_5, e_6, e_7\rangle\}$ and the traces σ_1 and σ_2 as defined below, we obtain

$$\begin{array}{rclcrcl} \sigma_1 & = & \langle \overbrace{e_0, e_2, e_3}^{\mathsf{tid}=1}, \overbrace{e_4, e_5, e_6}^{\mathsf{tid}=2}, \overbrace{e_8, e_9}^{\mathsf{tid}=1}\rangle & \qquad & \mathsf{macro}(\sigma_1) & = & \langle \overbrace{m_0, m_2}^{\mathsf{tid}=1}, \overbrace{m_3}^{\mathsf{tid}=2}, \overbrace{m_4}^{\mathsf{tid}=1}\rangle \\ \sigma_2 & = & \langle \underbrace{e_1, e_2}_{\mathsf{tid}=1}, \underbrace{e_5, e_6, e_7}_{\mathsf{tid}=2}, \underbrace{e_3, e_8, e_9}_{\mathsf{tid}=1}\rangle & & \mathsf{macro}(\sigma_2) & = & \langle \underbrace{m_1}_{\mathsf{tid}=1}, \underbrace{m_5}_{\mathsf{tid}=2}, \underbrace{m_2, m_4}_{\mathsf{tid}=1}\rangle \end{array} \tag{5.1}$$

This transformation reduces the number of events as well as the length of the traces while preserving the context switches which are necessary for understanding the cause of failures in concurrent programs.

However, transforming traces to macro traces hides information about the frequency of the original events. A mining algorithm applied to the macro traces will determine a support of one for m_3 and m_5, even though the events $\{e_5, e_6\} = \mathsf{events}(m_3) \cap \mathsf{events}(m_5)$ have a support of 2 in the original traces. While this problem can be amended by *refining* $\mathbb{M}$ by adding $m_6 = \langle e_5, e_6\rangle$, $m_7 = \langle e_4\rangle$, and $m_8 = \langle e_6\rangle$, for instance, this increases the length of the trace and the number of events, countering our original intention.

Instead, we introduce an abstraction function $\alpha : \mathbb{M} \to \mathbb{A}$ which maps macros to a set of abstract events $\mathbb{A}$ according to the events they share. The abstraction guarantees that if m_1 and m_2 share events, then $\alpha(m_1) = \alpha(m_2)$.

Definition 17 (Abstract events and traces)**.** *Let R be the relation defined as $R(m_1, m_2) \stackrel{\text{def}}{=} (\mathsf{events}(m_1) \cap \mathsf{events}(m_2) \neq \emptyset)$ and R^+ its transitive closure. We define $\alpha(m_i)$ to be $\{m_j \,|\, m_j \in \mathbb{M} \wedge R^+(m_i, m_j)\}$, and the set of abstract events $\mathbb{A}$ to be $\{\alpha(m) \,|\, m \in \mathbb{M}\}$. The abstraction of a macro trace $\mathsf{macro}(\sigma) = \langle m_1, m_2, \ldots, m_n\rangle$ is $\alpha(\mathsf{macro}(\sigma)) = \langle \alpha(m_1), \alpha(m_2), \ldots, \alpha(m_n)\rangle$.*

The concretization of an abstract trace $\langle a_1, a_2, \ldots, a_n\rangle$ is the set of macro traces $\gamma(\langle a_1, a_2, \ldots, a_n\rangle) \stackrel{\text{def}}{=} \{\langle m_1, \ldots, m_n\rangle \,|\, m_i \in a_i, 1 \leq i \leq n\}$. Therefore, we have $\mathsf{macro}(\sigma) \in \gamma(\alpha(\mathsf{macro}(\sigma)))$. Further, since for any $m_1, m_2 \in \mathbb{M}$ with $e \in \mathsf{events}(m_1)$ and $e \in \mathsf{events}(m_2)$ it holds that $\alpha(m_1) = \alpha(m_2) = a$ with $a \in \mathbb{A}$, it is guaranteed that $\mathsf{support}_\Sigma(e) \leq \mathsf{support}_{\alpha(\Sigma)}(a)$, where $\alpha(\Sigma) = \{\alpha(\mathsf{macro}(\sigma)) \,|\, \sigma \in \Sigma\}$. For the example above (5.1), we obtain $\alpha(m_i) = \{m_i\}$ for $i \in \{2, 4\}$, $\alpha(m_0) = \alpha(m_1) = \{m_0, m_1\}$, and $\alpha(m_3) = \alpha(m_5) = \{m_3, m_5\}$ (with $\mathsf{support}_{\alpha(\Sigma)}(\{m_3, m_5\}) = \mathsf{support}_\Sigma(e_5) = 2$).

5.2.2 Mining Patterns from Abstract Traces

As we will demonstrate in Section 5.4, abstraction significantly reduces the length of traces, thus facilitating sequential pattern mining. Since patterns mined from abstract traces contain abstract events, in order to be used for explaining concurrency bugs they have to be translated into the corresponding subsequences of the original traces. This translation is done by first concretizing them into sequences of macros which we refer to as *macro patterns*. The macros of each macro pattern are then concatenated to yield patterns which are subsequences of the original traces. We argue that the resulting set of patterns over-approximate the patterns of the corresponding original execution traces:

Lemma 5.2.1. *Let Σ be a set of execution traces, and let $\pi = \langle e_0, e_1 \dots e_k \rangle$ be a frequent pattern with $\mathsf{support}_\Sigma(\pi) = n$. Then there exists a frequent pattern $\langle a_0, \dots, a_l \rangle$ (where $l \leq k$) with support at least n in $\alpha(\Sigma)$ such that for each $j \in \{0..k\}$, we have $\exists m \,.\, e_j \in m \wedge \alpha(m) = a_{i_j}$ for $0 = i_0 \leq i_1 \leq \dots \leq i_k = l$.*

Lemma 5.2.1 follows from the fact that each e_j must be contained in some macro m and that $\mathsf{support}_\Sigma(e_j) \leq \mathsf{support}_{\alpha(\Sigma)}(\alpha(m))$. The pattern $\langle e_2, e_5, e_6, e_8, e_9 \rangle$ in the example above (5.1), for instance, corresponds to the abstract pattern $\langle \{m_0, m_1\}, \{m_3, m_5\}, \{m_4\} \rangle$ with support 2. Note that even though the abstract pattern is significantly shorter, the number of context switches is the same.

While our abstraction preserves the original patterns in the sense of Lemma 5.2.1, it may introduce spurious patterns. If we apply γ to concretize the abstract pattern from our example, we obtain four patterns $\langle m_0, m_3, m_4 \rangle$, $\langle m_0, m_5, m_4 \rangle$, $\langle m_1, m_3, m_4 \rangle$, and $\langle m_1, m_5, m_4 \rangle$. The patterns $\langle m_0, m_5, m_4 \rangle$ and $\langle m_1, m_3, m_4 \rangle$ are *spurious*, as the concatenations of their macros do not translate into valid subsequences of the traces σ_1 and σ_2.

Clearly, the supports of the original patterns are not preserved by abstraction. Following from Lemma 5.2.1, we only have $\mathsf{support}_\Sigma(\pi) \leq \mathsf{support}_{\alpha(\Sigma)}(\langle a_1, \dots, a_n \rangle)$ where π is a concrete pattern that is a subsequence of $m_1 \cdot \dots \cdot m_n$ with $m_i \in \gamma(a_i)$. Since the supports of the patterns obtained by the translation of abstract patterns are not precise, they are not necessarily closed according to definition of closed patterns in Section 5.1.4. Therefore, we only preserve the existence of patterns in $\mathrm{CS}_{\Sigma,\mathsf{min\_supp}}$ by mining $\mathrm{CS}_{\alpha(\Sigma),\mathsf{min\_supp}}$: for every pattern π in $\mathrm{CS}_{\Sigma,\mathsf{min\_supp}}$ there exists at least one macro pattern Π in $\gamma(\mathrm{CS}_{\alpha(\Sigma),\mathsf{min\_supp}})$ such that $\pi \sqsubseteq \mathsf{concat}(\Pi)$.

5.2.3 Deriving Macros from Traces

The precision of the approximation as well as the length of the trace is inherently tied to the choice of macros $\mathbb{M}$ for Σ. There is a tradeoff between precision and length: choosing longer subsequences as macros leads to shorter traces but also more intersections between macros.

In our algorithm, we start with macros of maximal length, splitting the traces in Σ into subsequences at the context switches. Subsequently, we iteratively refine the resulting set of macros by selecting the shortest macro m and splitting all macros that contain m as a substring. In the example in Section 5.2.1, we start with $\mathbb{M}_0 = \{m_0 \stackrel{\text{def}}{=} \langle e_0, e_2, e_3 \rangle, m_1 \stackrel{\text{def}}{=} \langle e_4, e_5, e_6 \rangle, m_2 \stackrel{\text{def}}{=} \langle e_8, e_9 \rangle, m_3 \stackrel{\text{def}}{=} \langle e_1, e_2 \rangle, m_4 \stackrel{\text{def}}{=} \langle e_5, e_6, e_7 \rangle, m_5 \stackrel{\text{def}}{=} \langle e_3, e_8, e_9 \rangle\}$. As m_2 is contained in m_5, we split m_5 into m_2 and $m_6 \stackrel{\text{def}}{=} \langle e_3 \rangle$ and replace it with m_6. The new macro is in turn contained in m_0, which gives rise to the macro $m_7 = \langle e_0, e_2 \rangle$. At this point, we have reached a fixed point, and the resulting set of macros corresponds to the choice of macros in our example.

For a fixed initial state, the execution traces frequently share a prefix (representing the initialization) and a suffix (the finalization). These are mapped to the same macro events

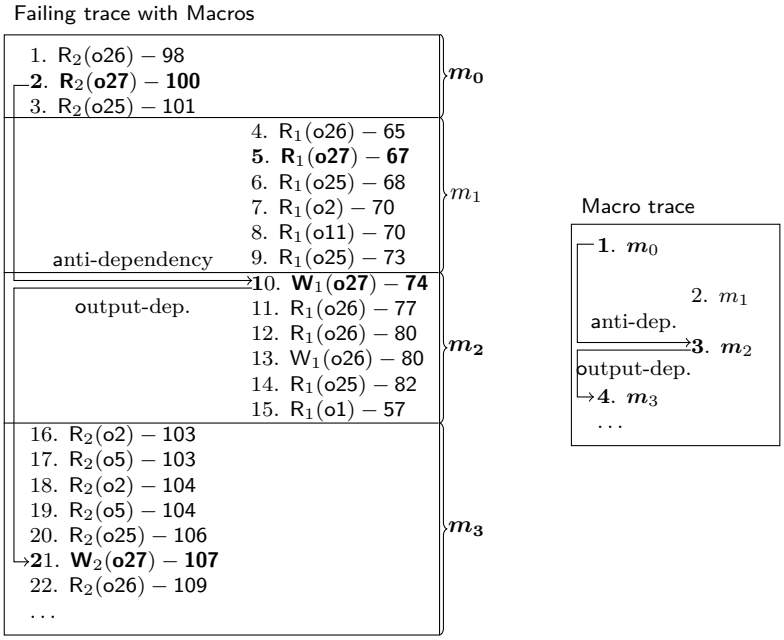


Figure 5.3: Bug explanation with macro pattern

by our heuristic. Since these macros occur at the beginning and the end of all passing as well as failing traces, we prune the traces accordingly and focus on the deviating substrings of the traces.

5.3 Bug Explanation Patterns at the Level of Macros

By transforming traces into macro traces and then abstracting them, we lift the Definition 13 of bug explanation patterns to sequences of macros, accordingly. We argue that similar to bug explanation patterns, macro patterns which are sequences of macros also reveal the problem but at a higher level. Since context switches are preserved inside a macro trace, a sequence of macros can expose unexpected or problematic context switches. Figure 5.3 shows the transformation of failing trace 2 in Figure 5.1 to a sequence of macros. The concurrency bug reflected by $\langle \mathsf{R}_2(\mathsf{o27}) - 100, \mathsf{W}_1(\mathsf{o27}) - 74, \mathsf{W}_2(\mathsf{o27}) - 107 \rangle$ similarly can be inferred from the sequence of macros $\langle m_0, m_2, m_3 \rangle$.

A macro pattern Π is a candidate bug explanation pattern if the following conditions are satisfied:

1. Π contains macros of at least two different threads. The rationale for this constraint is that we are exclusively interested in concurrency bugs.

2. For each macro in Π there is a data-dependency with at least one other macro in Π. We lift the data-dependencies introduced in Section 5.1.2 to macros as follows:

Two macros m_1 and m_2 are data-dependent iff there exist $e_1 \in \mathsf{events}(m_1)$ and $e_2 \in \mathsf{events}(m_2)$ such that e_1 and e_2 are related by dep.

3. Π is more frequent in the failing dataset than in the passing dataset (determined by the value of $\mathsf{rel\_supp}$).

Since there is empirical evidence that real world concurrency bugs involve only a small number of threads, context switches, and variables [LPSZ08, MQ07], we restrict our search to Πs with a limited number of context switches (at most 3). Accordingly, we mine patterns of length up to 4 from abstract traces (every abstract event corresponds to the events of one single thread). This heuristic limits the length of patterns and increases the scalability of our analysis significantly.

Although a sequence of macros such as Π explains the bug at a high-level, in the sense of Definition 13 there exists a bug pattern, for instance, $\pi = \langle e_1, e_2, \ldots, e_m \rangle$ such that $\pi \sqsubseteq \mathsf{concat}(\Pi)$. For example, $\langle \mathsf{R}_2(\mathsf{o27}) - 100, \mathsf{W}_1(\mathsf{o27}) - 74, \mathsf{W}_2(\mathsf{o27}) - 107 \rangle$ in Figure 5.3 is a subsequence of $\mathsf{concat}(\langle m_0, m_2, m_3 \rangle) = m_0 \cdot m_2 \cdot m_3$.

In other words, Π provides the context in which π occurs in a failing trace. Since π does not occur necessarily in the same context in different traces, in general there are a number of macro patterns $\Pi_1, \Pi_2, \ldots, \Pi_n$ which contain π as a subsequence. Consequently, all these macro patterns reflect the same problem.

5.3.1 Algorithm

Before discussing the individual steps of our bug explanation technique (Algorithm 5.2), we provide a brief outline of the sequence mining algorithm it relies on. For mining the closed set of patterns from the abstract traces, we apply Algorithm 5.1, a mining algorithm similar to PrefixSpan [PHMA+01]. The algorithm is based on the *Apriori* property, which states that any super-sequence of a non-frequent sequence cannot be frequent. Therefore, the algorithm starts by finding frequent single events which are then incrementally extended to frequent patterns. Procedure MineClosedPatterns calls the procedure MineRecursive to recursively extend frequent patterns. In each recursive call, procedure MineRecursive first computes all frequent events in the input dataset Σ (line 11). In the first iteration, this dataset is equal to the input dataset of MineClosedPatterns. It then uses these frequent events to extend *pat*, the last mined frequent pattern (line 13). Since patterns are extended by adding only one frequent event e to *pat*, the input dataset is shrunk by *projection* (line 15), which shortens the sequences by removing their prefixes containing the first occurrence of e. This is due to the fact that these prefixes do not contain any instances of patterns longer than the extended pattern *nextPat*, and they can be safely removed from the sequences. The projected dataset $new\Sigma$ is then used in the subsequent call for growing *nextPat*.

The check whether a pattern is closed is done at line 14 by calling the procedure UpdateClosed. We mine frequent patterns up to the length determined by parameter

Algorithm 5.1 Mining closed patterns

```
1: procedure MineClosedPatterns(Σ, min_supp, max_pattern_len)
2:     closed = {}
3:     pat = {}
4:     MineRecursive(pat, Σ, min_supp, max_pattern_len, closed)
5:     return closed
6: end procedure

7: procedure MineRecursive(pat, Σ, min_supp, max_pattern_len, closed)
8:     if |pat| ≥ max_pattern_len then
9:         return
10:    end if
11:    Freq = {e|e ∈ events(Σ) ∧ support_Σ(e) ≥ min_supp}
12:    for every e in Freq do
13:        nextPat = pat + e
14:        UpdateClosed(nextPat, closed)
15:        newΣ = prj(Σ)_e
16:        MineRecursive(nextPat, newΣ, min_supp, max_pattern_len, closed)
17:    end for
18: end procedure
```

Algorithm 5.2 Steps of the bug explanation method

Input: $\Sigma_F, \Sigma_P, \text{min\_supp}$
Output: *bug_candidate_patterns*

```
1: ⟨α(Σ_F), α(Σ_P)⟩ ← AbstractTraces(Σ_F, Σ_P)
2: CS_α(Σ_F),min_supp ← MineClosedPatterns(α(Σ_F), min_supp, 4)
3: AbsPat ← FilterPatterns_withNoContextSwitch(CS_α(Σ_F),min_supp)
4: MacroPat_0 ← ConcretizeAbstractPatterns(AbsPat)
5: MacroPat_1 ← FilterSpuriousPatterns(MacroPat_0, macro(Σ_F))
6: MacroPat_2 ← FilterPatterns_withNoDataDep(MacroPat_1, macro(Σ_F))
7: RelSup ← ComputeRelSupp(MacroPat_2, macro(Σ_P), macro(Σ_F))
8: bug_candidate_patterns ← Rank_GroupPatterns(MacroPat_2, RelSup)
```

max_pattern_len (line 8). As discussed at the beginning of this section, this parameter is set to the heuristically chosen value of 4.

Algorithm 5.1 is applied as the second step of our method for generating bug explanation patterns (shown in Algorithm 5.2). The mining algorithm computes the closed patterns of length at most 4 that are frequent in the abstracted failing dataset $\alpha(\Sigma_F)$, which is constructed in the first step.

Subsequently, we filter abstract patterns that do not contain context switches in step 3 of Algorithm 5.2 (as motivated in Section 5.3). The resulting patterns *AbsPat* may still

contain spurious patterns which have no counterpart in the concrete dataset. In order to filter spurious patterns, the abstract patterns need to be mapped to macro patterns $MacroPat_0$, which is done in step 4.

Steps 5 through 7 perform the filtering steps described in Section 5.3: step 5 eliminates spurious patterns that do not occur in the original set of failing traces, step 6 eliminates patterns whose events are not related by the dependency relation dep, as required by Definition 13, and step 7 computes the relative support of the remaining patterns. From these patterns, we only keep those whose rel_supp is greater than 0.5 (Definition 13). Since there may be several patterns with the same rel_supp, at step 8, we group the patterns according to the value of relative support and the set of data-dependencies they contain. Therefore, patterns inside one group have the same rel_supp and set of data-dependencies. Intuitively, they refer to the same bug. Finally, we rank these groups of patterns according to rel_supp. Groups with maximum rel_supp are ranked highest in the final result set and consequently inspected first by the user.

The filtering operations of steps 5 through 7 require inspection of *original* execution traces. For this purpose, we can use either the concrete traces or the macro traces as a reference. Accordingly, we have the following two options:

- Mapping macro patterns to original traces, providing the original datasets Σ_F and Σ_P (instead of $\mathsf{macro}(\Sigma_F)$ and $\mathsf{macro}(\Sigma_P)$) as inputs to the procedures of steps 5-7.
- Mapping macro patterns to macro traces instead of original traces and providing $\mathsf{macro}(\Sigma_F)$ and $\mathsf{macro}(\Sigma_P)$ as inputs to the procedures of steps 5-7.

Since macro traces are significantly shorter than the original traces, the second option results in orders of magnitude speedup in run time. The first option, however, yields a precise value of the (relative) supports for the macro patterns, while the second option results in an *under*-approximation of the supports. This is due to the fact that by computing only the instances (Definition 14) of a macro pattern inside a *macro* trace (rather than the corresponding original trace), we exclude instances of the pattern in which the events of one macro do not occur next to each other inside an original trace. For example, for $m_0 \stackrel{\text{def}}{=} \langle e_1, e_2, e_3 \rangle, m_1 \stackrel{\text{def}}{=} \langle e_1, e_3 \rangle, m_2 \stackrel{\text{def}}{=} \langle e_4, e_5 \rangle$, the trace $\sigma = \langle e_1, e_2, e_3, e_4, e_5 \rangle$, and the macro pattern $\Pi = \langle m_1, m_2 \rangle$, we have $\Pi \not\sqsubseteq \mathsf{macro}(\sigma)$ although $(\mathsf{concat}(\Pi) = \langle e_1, e_3, e_4, e_5 \rangle) \sqsubseteq \sigma$. The reason is that in the instance of $\mathsf{concat}(\Pi)$ in σ (cf. Definition 14), e_1 and e_3 do not occur next to each other.

In the method of [TBWW14], we used the first option in the implementation of the method while in the method of this paper we used the second option. Therefore, we improved performance of the method at the cost of precision of the supports of macro patterns. Since the ratio between the support of patterns in the failing and passing datasets is taken into account, the under-approximation of the supports does not affect the effectiveness of the method as we will see in Section 5.4. We argue that the instances of macro patterns we do not take into account using the modified method are insignificant

for the purpose of bug explanation. This is because corresponding to every bug pattern π there exists at least one macro pattern Π such that $\pi \sqsubseteq \mathsf{concat}(\Pi)$. Since macro patterns are mined from macro traces, they necessarily occur as a subsequence of at least one macro trace. In other words, macro patterns have an instance inside at least one macro trace. Therefore, the modified method is capable of capturing them.

Parameters of the method. For understanding the cause of a failure, the final result-set *bug_candidate_patterns* needs to be inspected by the programmer. In this result set, patterns ranked highest are inspected first. Intuitively, they are most likely to be indicative of a bug. It must be noted that our method is not supposed to be complete, and we use the method as part of an iterative debugging process. Therefore, as soon as the user understands the cause of failure, he will try to remove the bug. In case the program still contains bugs after being modified, the user will apply the method again. In our experiments, in every case study the first pattern in *bug_candidate_patterns* was indicative of the single bug in the program, hence freeing the user from the obligation to inspect all patterns in the list or multiple applications of the method.

The bug explanation patterns are evaluated by the user. If the method does not generate useful patterns (according to user verdict) in the first iteration, there are different parameters which can be tuned to generate a new set of patterns. These parameters include min_supp, max_pattern_len, Σ_F and Σ_P. In the experimental result section, we analyze the effect of min_supp and traces with bounded number of context switches on the output of method.

5.4 Experimental Evaluation

To evaluate our approach, we present nine case studies which are listed in Table 5.2 (6 of them are taken from [LC09]). The programs are C/C++ codes which belong to three different categories: full applications, bug kernels and synthetic buggy code. The bug kernels were extracted from Mozilla and Apache. They are 135-300 lines of code programs which capture the essence of bugs reported in Mozilla and Apache. Synthetic examples were created to cover a specific bug category. *bzip2smp* is a real multithreaded application which uses multiple threads to speed up the compression of a file. Since the original version taken from [bzi] does not contain a bug, we injected an atomicity violation bug in the code.

We generate execution traces using the concurrency testing tool Inspect [YCGK07], which systematically explores interleavings for a fixed program input. The generated traces are then classified as failing and passing traces with respect to the violation of a property of interest. We implemented our mining algorithm in C#. All experiments were performed on a 2.60 GHz PC with 8 GB RAM running 64-bit Windows 7.

Our experiments were designed to answer three research questions:

Prog. Category	Name	App. Version	Bug Type	LOC	Threads
Synthetic	BankAccount	n/a	SAV	140	3
	CircularListRace	n/a	SAV	130	3
	WrongAccessOrder	n/a	OV	112	3
Bug Kernel	Apache-25520(Log)	Apache-2.0.48	SAV	135	4
	Moz-jsClrMsgPane	Mozilla	MAV	290	3
	Moz-jsStr	Mozilla-0.9	MAV	242	3
	Moz-jsInterp	Mozilla-0.8	MAV	206	3
	Moz-txtFrame	Mozilla-0.9	MAV	230	3
Full App.	bzip2smp	bzip2smp 1.0	MAV	6400	3

SAV: Single-variable Atomicity Violation MAV: Multi-variable Atomicity Violation
OV: Order Violation

Table 5.2: Characteristics of the case studies

- Can our abstraction technique efficiently reduce the length of the traces, so that mining sequential patterns becomes tractable? (Section 5.4.1)
- Do the generated bug explanation patterns accurately reveal the problematic context switches which caused the failure in a concurrent program? (Sections 5.4.2, 5.4.3)
- To which extent does the quality of the method depend on the given datasets? (Sections 5.4.5, 5.4.6)

5.4.1 Length Reduction by Abstraction

First, we evaluate the efficacy of our abstraction technique. In Table 5.3, for every case study the number of traces inside the failing and passing datasets and their average lengths are given in columns 2, 3 and 4, respectively. We use the case studies indicated by "*" to generate long traces by increasing the size of the data structures in the corresponding original case studies. For the traces in this table, the last column shows the average length reduction (up to 99%) achieved by means of abstraction. For the given case studies, the length is reduced by 91% on average.

State-of-the-art sequential pattern mining algorithms are typically applicable to sequences of length less than 100 [YHA03, ME10]. Therefore, reduction of the original traces is crucial. For five case studies (corresponding to rows 1,2,3,8,9,10 in Table 5.3), we used an exhaustive set of interleavings – i.e., all execution traces Inspect was able to generate. For *WrongAccessOrder* and *Apache-25520(Log)*, we took the first 100 failing and 100 passing traces from the sets of 1427 and 32930 traces we were able to generate. For *Moz-jsClrMsgPane* and *Apache-25520(Log)**, failing and passing traces are chosen from the first 820 and 702 traces generated by Inspect. For *bzip2smp*, we generated 220 traces using Inspect (the first 200 of which were passing) and then chose the first 20 failing and 20 passing traces from them. In Section 5.4.6, we study the effect of input datasets by randomly choosing 100 failing and 100 passing traces from the set of available traces.

Prog. Name	$\|\Sigma_F\|$	$\|\Sigma_P\|$	Avg. Trace Len.	Avg. Abst. Len.	Avg. Len. Red.
BankAccount	40	5	178	13	93%
CircularListRace	64	6	187	9	95%
CircularListRace*	64	6	13,122	9	99%
WrongAccessOrder	100	100	73	19	74%
Apache-25520(Log)	100	100	115	15	87%
Apache-25520(Log)*	675	27	4,219	14	99%
Moz-jsClrMsgPane	775	45	7,144	15	99%
Moz-jsStr	70	66	407	18	95%
Moz-jsInterp	610	251	433	89	79%
Moz-txtFrame	99	91	409	57	86%
bzip2smp	20	20	12,997	13	99%

Table 5.3: Length reduction results by abstracting the traces

5.4.2 Effectiveness of the Method

In this section, we report quantitatively on the number of the final patterns generated by the method (in the worst case the user has to inspect all of them). We also discuss the effectiveness of the mined patterns in understanding concurrency bugs. The results of mining bug explanation patterns for the given programs and traces are provided in Figure 5.4. The number of the generated patterns depends on the given value of the minimum support threshold (Section 5.1.4). Since lower thresholds yield more patterns, in the experiments we start from the maximum value of 100% and decrease it only if it is not sufficient for generating at least one useful pattern which accurately reveals the cause of the failure. The horizontal axis labeled min_supp in Figure 5.4 shows the support threshold values used in the experiments. For all case studies except *Moz-txtFrame*, the maximum value of 100% is sufficient to obtain at least one useful pattern. For *Moz-txtFrame*, we had to gradually decrease the threshold to 90% to find at least one explanation.

The vertical axis shows the number of patterns (on a logarithmic scale) generated after different steps of Algorithm 5.2. For every case study, for the given value of min_supp, three columns from left to right, respectively, show the number of resulting abstract patterns (step 2), the number of feasible or non-spurious patterns (step 5) and the number of patterns remaining after removing patterns which do not satisfy the data-dependency constraints (step 6). The fourth column from left shows the number of patterns with maximum relative support of 1 (which only occur in the failing dataset). Although step 7 of the algorithm computes the patterns whose rel_supp is greater than 0.5 (which only frequent in the failing dataset), since for most case studies the algorithm produced several patterns with rel_supp = 1, only the number of these patterns are reported in Figure 5.4. The rightmost column for every case study in Figure 5.4 shows the number of groups that these patterns can be divided into according to the set of data-dependencies they contain. Since there are several of these groups, we sort them in descending order according to the number of data-dependencies. Therefore, in the final result set a group of patterns

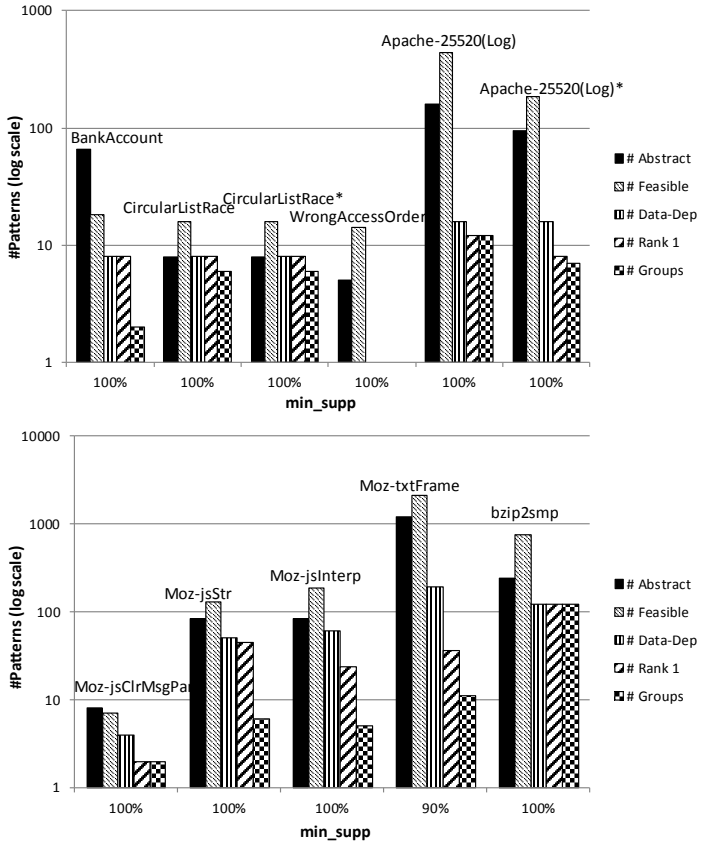


Figure 5.4: Mining results

with the highest value of relative support and maximum number of data-dependencies appears at the top.

The patterns at the top of the list in the final result are inspected first by the user in order to understand a bug. For the case study *WrongAccessOrder* since #Data-Dep #Rank 1 and #Groups are all 1, the corresponding columns in Figure 5.4 are not drawn due to the log scale of vertical axis. As the last column in Figure 5.4 shows, the resulting number of the groups for most case studies is less than 10. (The relatively large number of final groups for *bzip2smp* case study can be an effect of choosing a relatively small set of input traces.)

Mining of abstract patterns (step 2) takes around 87ms on average. With an average runtime of 27s, the post-processing after mining (step 3-8) is the computationally most expensive step, but is very effective in eliminating irrelevant patterns.

We verified manually that all groups with the relative support of 1 (Figure 5.4) are an adequate explanation of at least one concurrency bug in the corresponding program. In the following, we explain for each case study how the inspection of only a single pattern from these groups can expose the bug. These patterns are given in Figure 5.5. For each

case study, the given pattern belongs to a group of patterns which appeared at the top of the list in the final result set, hence inspected first by the user. In this figure, we only show the ids of the events and the data-dependencies relevant for understanding the bugs. Macros are separated by extra spaces between the corresponding events. It must be noted that the events inside a macro occur consecutively inside the traces while between the macros there can be a context switch. As we will explain in the following, from the data-dependencies between the macros we can infer problematic context switches between the threads.

According to the commonly used classification, we have 3 different types of concurrency bugs in our case studies, namely single- and multi-variable atomicity violations, and order violations.

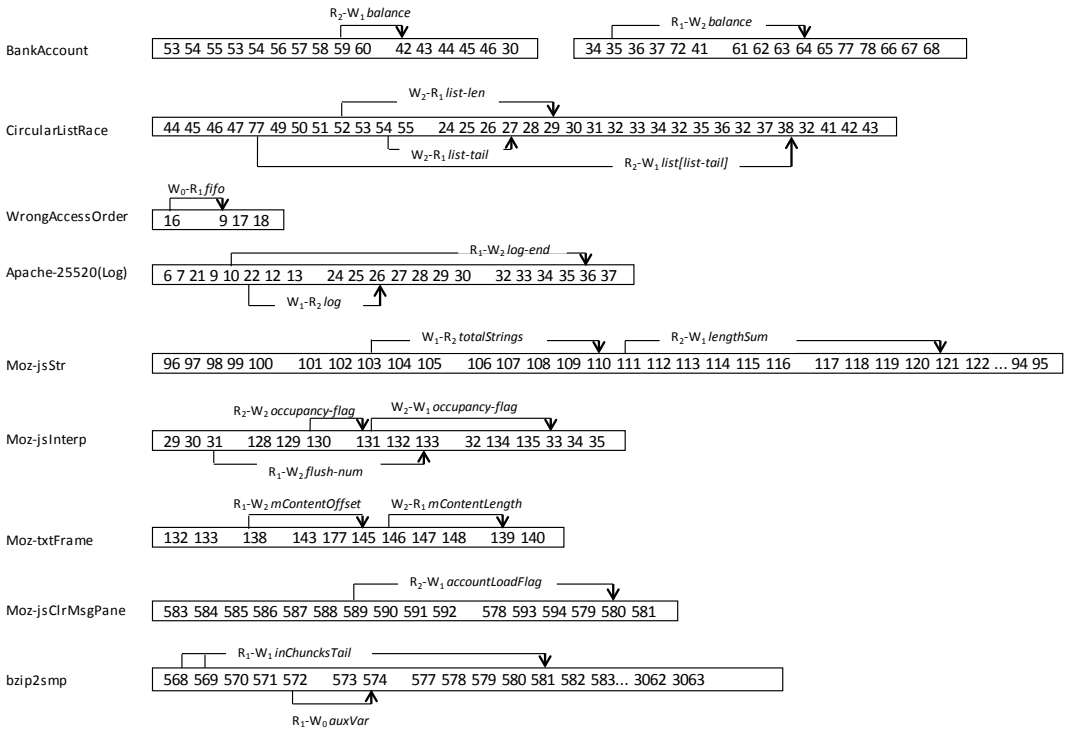


Figure 5.5: Bug explanation patterns—case studies

Single-variable Atomicity Violation

Bank Account. The update of the shared variable balance in Figure 5.1 in Section 5.1.3 involves a *read* as well as a *write* access that are not located in the same critical region. Accordingly, a context switch may result in writing a stale value of balance. In Figure 5.5, we provide two patterns for *BankAccount*, each of which contains two macro events. Figure 5.6 shows these patterns by mapping the ids to the corresponding read/write events. From the anti-dependency ($R_2 - W_1$ balance) in the left pattern, we infer an

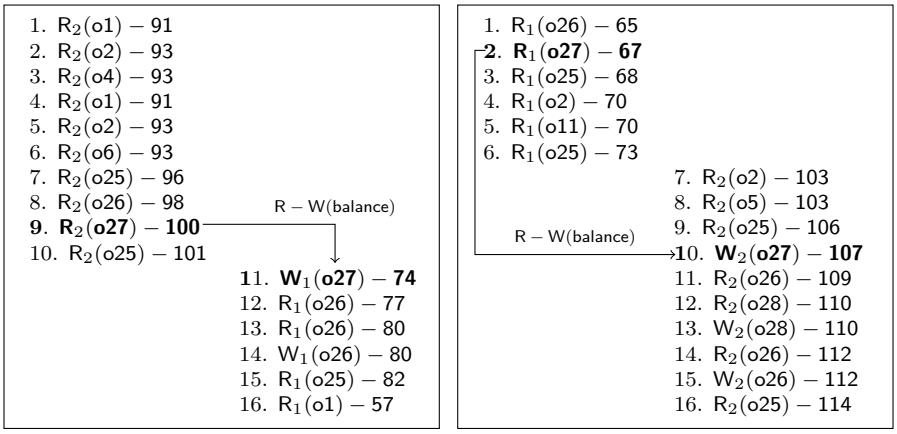


Figure 5.6: Expansion of bug explanation patterns—bank account

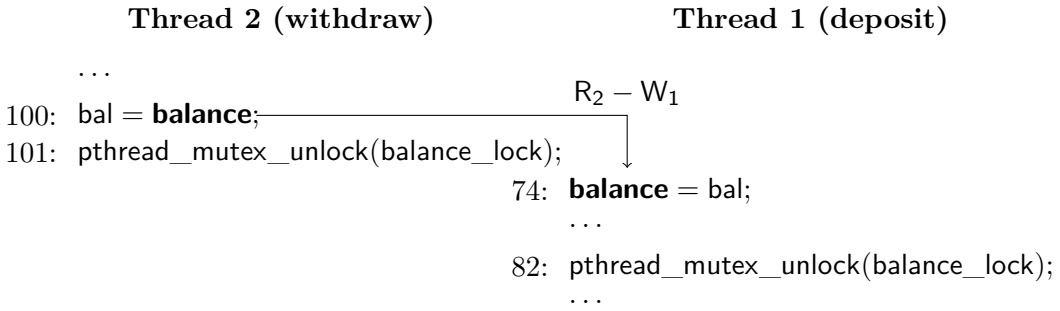


Figure 5.7: Mapping of bug pattern to source code

atomicity violation in the code executed by thread 2, since a context switch occurs after R_2(balance), consequently it is not followed by the corresponding W_2(balance). Similarly, from the anti-dependency $R_1 - W_2$ balance in the right pattern we infer the same problem in the code executed by the thread 1. Since the events of these patterns include the location in the source code, we can easily map them back to the corresponding lines of source code. Figure 5.7 shows part of the mapping of the left pattern to the source code. Patterns are visualized in this way and given to the user for inspection.

Circular List Race, Circular List Race*. This program removes elements from the end of a list and adds them to the beginning using the methods getFromTail and addAtHead, respectively. The update is expected to be atomic, but since the calls are not located in the same critical region, two simultaneous updates can result in an incorrectly ordered list if a context switch occurs. The first and the second macros of the pattern in Figure 5.5 correspond to the events issued by the execution of methods getFromTail by thread 2 and addAtHead by thread 1, respectively. Figure 5.8 shows the pattern by mapping the ids to the corresponding read/write events. From the given data-dependencies it can be inferred that these two calls occur consecutively during the

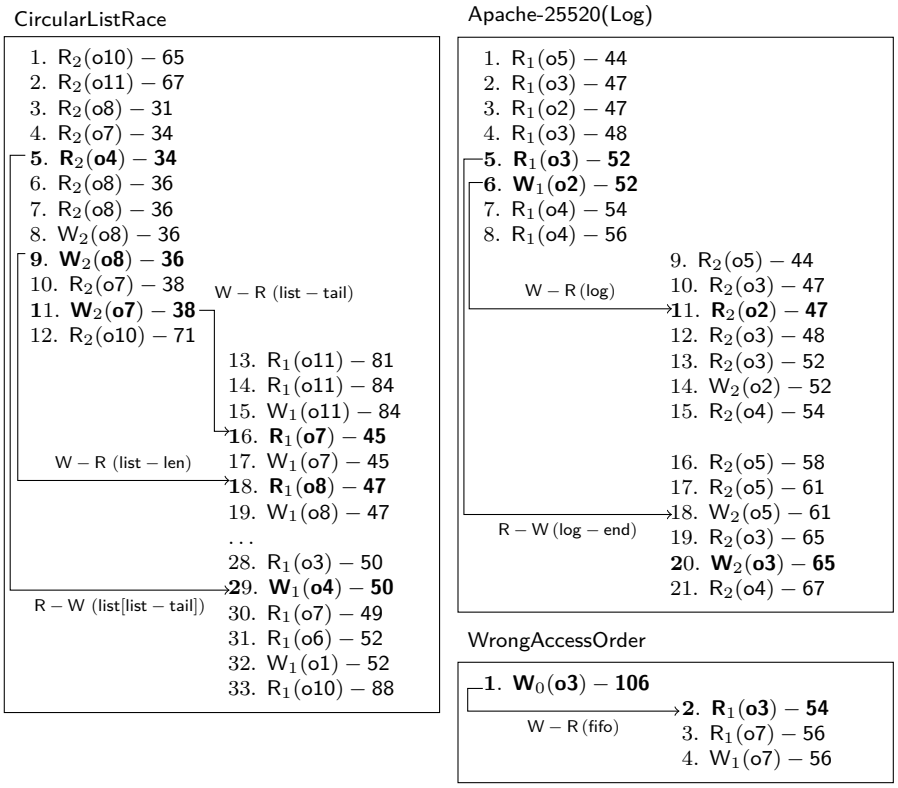


Figure 5.8: Expansion of bug explanation patterns—cont.

program execution, thus revealing the atomicity violation. This is due to the fact that the call of getFromTail by thread 2 should be followed by the call of addAtHead from the same thread.

Apache-25520(Log), Apache-25520(Log)*. In this bug kernel, Apache modifies a data-structure log by appending an element and subsequently updating a pointer to the log. Since these two actions are not protected by a lock, the log can be corrupted if a context switch occurs. The first macro of the pattern in Figure 5.5 (Figure 5.8) reflects thread 1 appending an element to log. The second and third macros correspond to thread 2 appending an element and updating the pointer, respectively. The dependencies imply that the modification by thread 1 is not followed by the corresponding update of the pointer.

Order Violation

Wrong Access Order. In this program, the main thread spawns two threads, consumer and output, but it only joins output. After joining output, the main thread frees the shared data-structure which may be accessed by consumer which has not exited yet. The

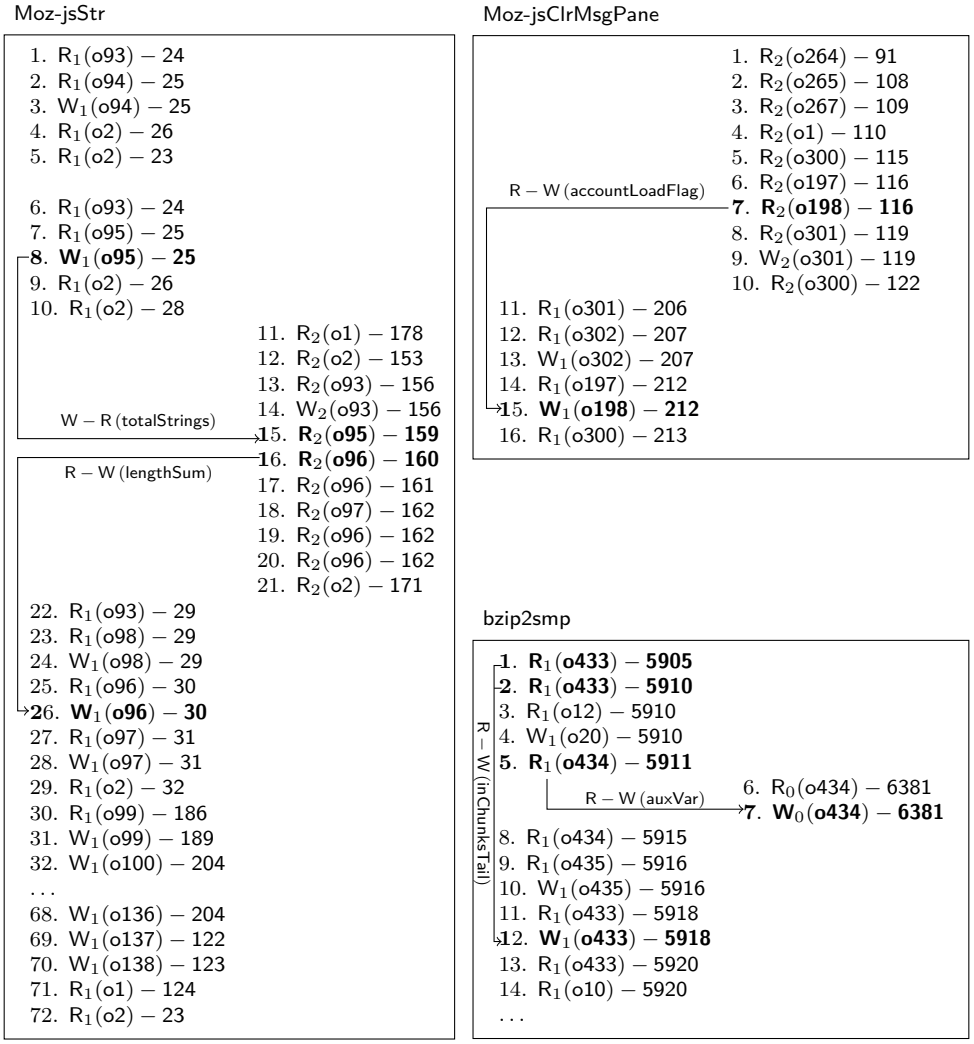


Figure 5.9: Expansion of bug explanation patterns—cont.

flow-dependency between the two macros of the pattern in Figure 5.5 (Figure 5.8) implies the wrong order in accessing the shared data-structure.

Multi-variable Atomicity Violation

Moz-jsStr. In this bug kernel, the cumulative length and the total number of strings stored in a shared cache data-structure are stored in two variables named lengthSum and totalStrings. These variables are updated non-atomically, resulting in an inconsistency. The pattern and the data-dependencies in Figure 5.5 (Figure 5.9) reveal this atomicity violation: the values of totalStrings and lengthSum read by thread 2 are inconsistent due to a context switch that occurs between the updates of these two variables by thread 1.

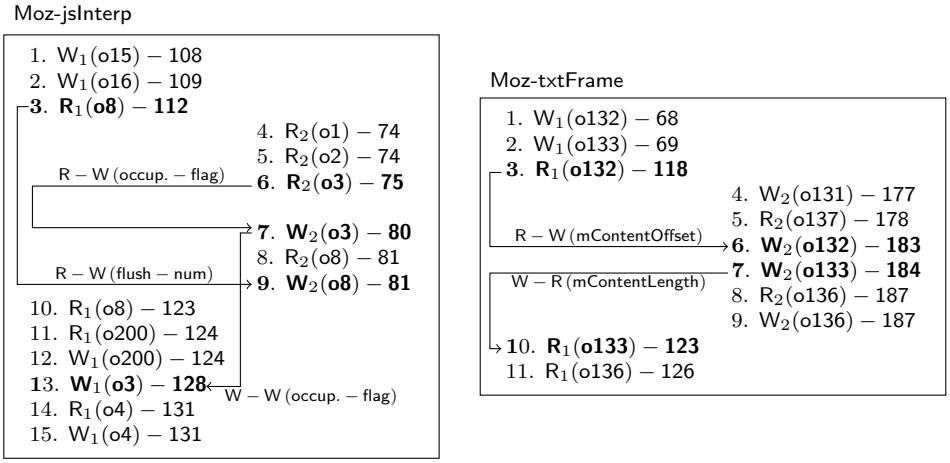


Figure 5.10: Expansion of bug explanation patterns—cont.

Moz-jsInterp. This bug kernel contains a non-atomic update to a shared data-structure Cache and a corresponding occupancy flag, resulting in an inconsistency between these objects. The first and last macro-events of the pattern in Figure 5.5 (Figure 5.10) correspond to populating Cache and updating the occupancy flag by thread 1, respectively. The other two macros show the flush of *Cache* content and the resetting of occupancy flag by thread 2. The given data-dependencies suggest the two actions of thread 1 are interrupted by thread 2 which reads an inconsistent flag.

Moz-txtFrame. The pattern and data-dependencies of this case study in Figure 5.5 (Figure 5.10) reflect a non-atomic update to the two fields mContentOffset and mContentLength, which causes the values of these fields to be inconsistent: the values of these variables read by thread 1 in the second and forth macros are inconsistent due to the updates done by thread 2 in the third macro.

Moz-jsClrMsgPane. In this bug kernel, there is a flag named accountLoadFlag which is set to true when the content of the data-structure account is loaded in to the corresponding window frame. Since the second macro of the given pattern for this case study in Figure 5.5 (Figure 5.9) contains only the update of accountLoadFlag, it can be inferred that the update of the flag and loading of account are not done atomically which results in an inconsistency between these two variables.

bzip2smp. In this multithreaded application, updates of the buffer inChunks and its pointer inChunksTail are not done in the same critical section. Therefore, occurrence of a context switch between these two updates results in an inconsistency between the buffer and pointer. The bug pattern of this application in Figure 5.5 (Figure 5.9) reflects the occurrence of a context switch between the updates of the buffer (first macro) and the pointer (third macro).

5.4.3 User Case Study Evaluation

To evaluate the effectiveness of bug explanation patterns in facilitating debugging concurrent programs, we ran a user case study with a group of undergraduate computer science students at Vienna University of Technology (TU Wien). We had two groups containing 16 students each. We gave one group the bug explanation patterns of three case studies namely *WrongAccessOrder*, *Moz-jsInterp* and *Moz-jsStr*. We used the other one as the control group given only the source codes of the case studies. We refer to the former as "M" (for mining) and latter as "S" (for source code). We asked the students to find the corresponding concurrency bugs either by reading the source code (group "S") or by inspecting given patterns (group "M"). For *WrongAccessOrder*, *Moz-jsInterp*, the violated assertions were specified in the source code and for *Moz-jsStr* a failing test case was given in addition to source code. Table 5.4 summarizes the results. This table for every programming task shows the number of the students in each group which were able to find the concurrency bugs correctly (columns 2, 3) and the amount of time on average that they spent on each task (columns 4, 5). As we can see, students in the group "M" by using the bug patterns were on average 5 minutes faster in finding the bugs. However, for two tasks, a larger number of students in group "S" were able to locate the bug correctly. We attribute this to the fact that the students of group "S" had more programming experiences according to their self-reported programming experience level. In order to verify this conjecture, we divided the students of each group into three subgroups of novice, average, and expert programmers according to their self-reported level of programming experience. Since the majority of the students were average programmers (11 in group "M" and 9 in group "S"), we only compared the performance of the average subgroups. These programmers performed better in group "M". On average 74% and 72% of them correctly found the bugs in groups "M" and "S", respectively. However, the average subgroup of "M" by spending 41 minutes on average were around 11 minutes faster than similar subgroup in "S". According to the feedback of the average programmers in group "M", the given patterns were *helpful* in finding the bugs. They found the given tasks at the medium level of difficulty.

Prog. Name	#Correct Ans.		Avg. Time (min.)	
	M	**S**	**M**	**S**
WrongAccessOrder	9	8	13	19.5
Moz-jsInterp	10	13	15	18
Moz-jsStr	10	13	9	14
Total Avg.	9.7	11.3	12	17

M: Mining group **S**: Source code group

Table 5.4: User case study results

5.4.4 Comparison with our Previous Method in [TBWW14]

As discussed in Section 5.3.1, using $\mathsf{macro}(\Sigma_F)$ and $\mathsf{macro}(\Sigma_P)$ instead of original datasets may result in pattern loss at step 5 and an under-approximation of supports at step 7 of Algorithm 5.2. The diagrams in Figure 5.11 show a comparison of the difference between the number of patterns generated at steps 5-8 of Algorithm 5.2 by method of this chapter (current) and our method presented in [TBWW14] (previous). We observed only a slight change between the outputs of the two methods in every step. In particular, the number of groups of patterns (step 8) is quite similar for all case studies.

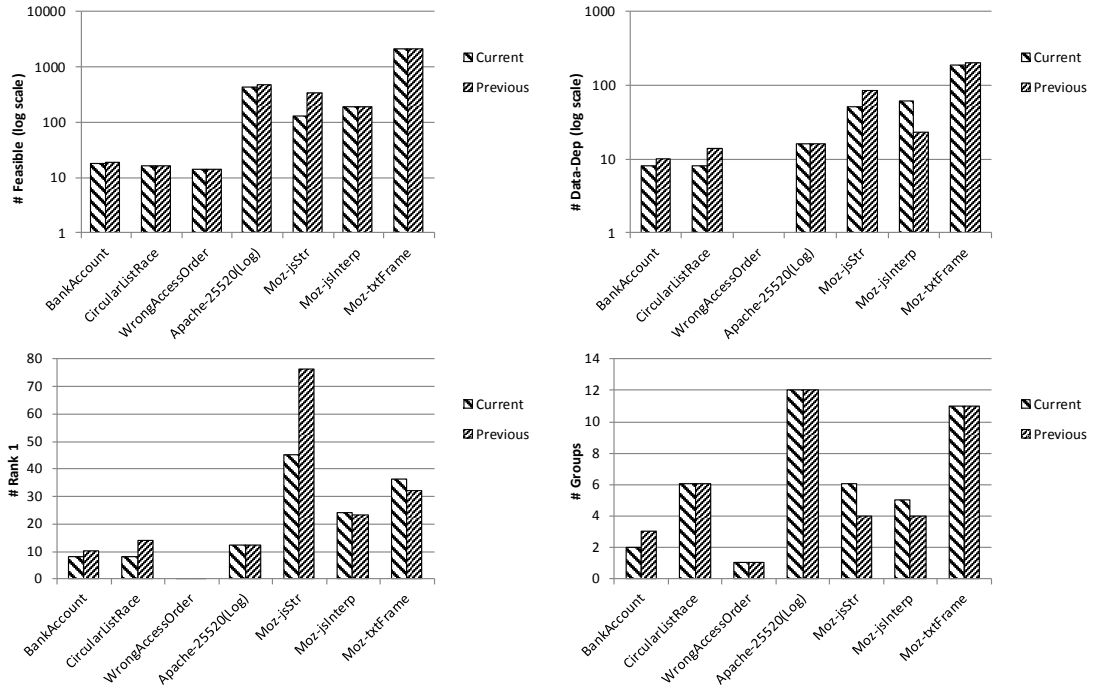


Figure 5.11: Comparison between current and previous methods

Considering the effectiveness of the patterns computed by the current method (as we discussed in the previous section), we came to the conclusion that the slight change in the number of patterns has not affected the quality of the final result-set or effectiveness of the current method. Moreover, our modification of the algorithm resulted in a speed up in running time as Table 5.5 shows. We use "–" to denote that post-processing step did not finish within 24 hours.

5.4.5 Datasets with Context-switch bounded Traces

In this section, we study the effect of Σ_F and Σ_P on the output of the method. As we have seen in Section 5.4.1, the datasets of some of our case studies do not contain all the executions that can be generated by Inspect. In this and next section, we show that the

Program	Mining abst. patt. time	Post-processing time	
		Previous	Current
BankAccount	30 ms	141 ms	38 ms
CircularListRace	26 ms	2269 ms	45 ms
CircularListRace*	28 ms	–	333 ms
WrongAccessOrder	32 ms	72 ms	40 ms
Apache-25520(Log)	55 ms	1207 ms	240 ms
Apache-25520(Log)*	117 ms	5745 ms	491 ms
Moz-jsClrMsgPane	70 ms	–	941 ms
Moz-jsStr	29 ms	86.573 s	163 ms
Moz-jsInterp	257 ms	1612.785 s	3200 ms
Moz-txtFrame	266 ms	29.929 s	6058 ms
bzip2smp	46 ms	–	280.595 s
Average	87 ms	–	27 s

Table 5.5: Efficiency of the previous and current method

Program	#context-switch		original		context-switch bound	
	max	bound	$\|\Sigma_F\|$	$\|\Sigma_P\|$	$\|\Sigma_F\|$	$\|\Sigma_P\|$
BankAccount	4	3,2	40	5	19,5	5,5
CircularListRace	7	6,5,4,3	64	6	62,56,38,20	6,6,6,6
WrongAccessOrder	11	6,5,4	100	100	11,5,1	49,18,7
Apache-25520(Log)	10	5,4,3	100	100	33,10,2	63,36,13
Moz-jsClrMsgPane	8	6,5,4,3	775	45	516,278,102,27	45,45,45,19
Moz-jsStr	5	4,3	70	66	15,5	30,12
Moz-jsInterp	4	3,2	610	251	59,20	61,22
Moz-txtFrame	5	4,3	99	91	18,6	36,14

Table 5.6: Datasets with context switch bounded traces

method does not rely on an exhaustive enumeration of failing and passing interleavings in order to compute patterns which are indicative of bugs. By bounding the number of context switches inside the traces, we generate different passing and failing datasets. The number of traces in these datasets for each case study is given in Table 5.6. In this table, we can see how the size of Σ_F and Σ_P is reduced by bounding the number of context switches using different bounds. For comparison, in Table 5.6 the size of datasets generated without a bound on the number of context switches (column 3 with the header "original") is also given. The maximum number of context switches in these datasets is also given in column 2 with the header named *max*. They are the same as the datasets in Table 5.3 and were used in the experiments of Section 5.4.2. The diagrams in Figure 5.12 show the effect of datasets containing context switch bounded traces on the number of

patterns generated at different steps of Algorithm 5.2. Although datasets with lower bounds contain fewer traces, in most case studies there is only a small change in the number of the generated patterns. Especially the last two bars from the right (#Rank 1 and #Groups) corresponding to the number of patterns with relative support of 1 and the number of groups of these patterns in most diagrams are very similar.

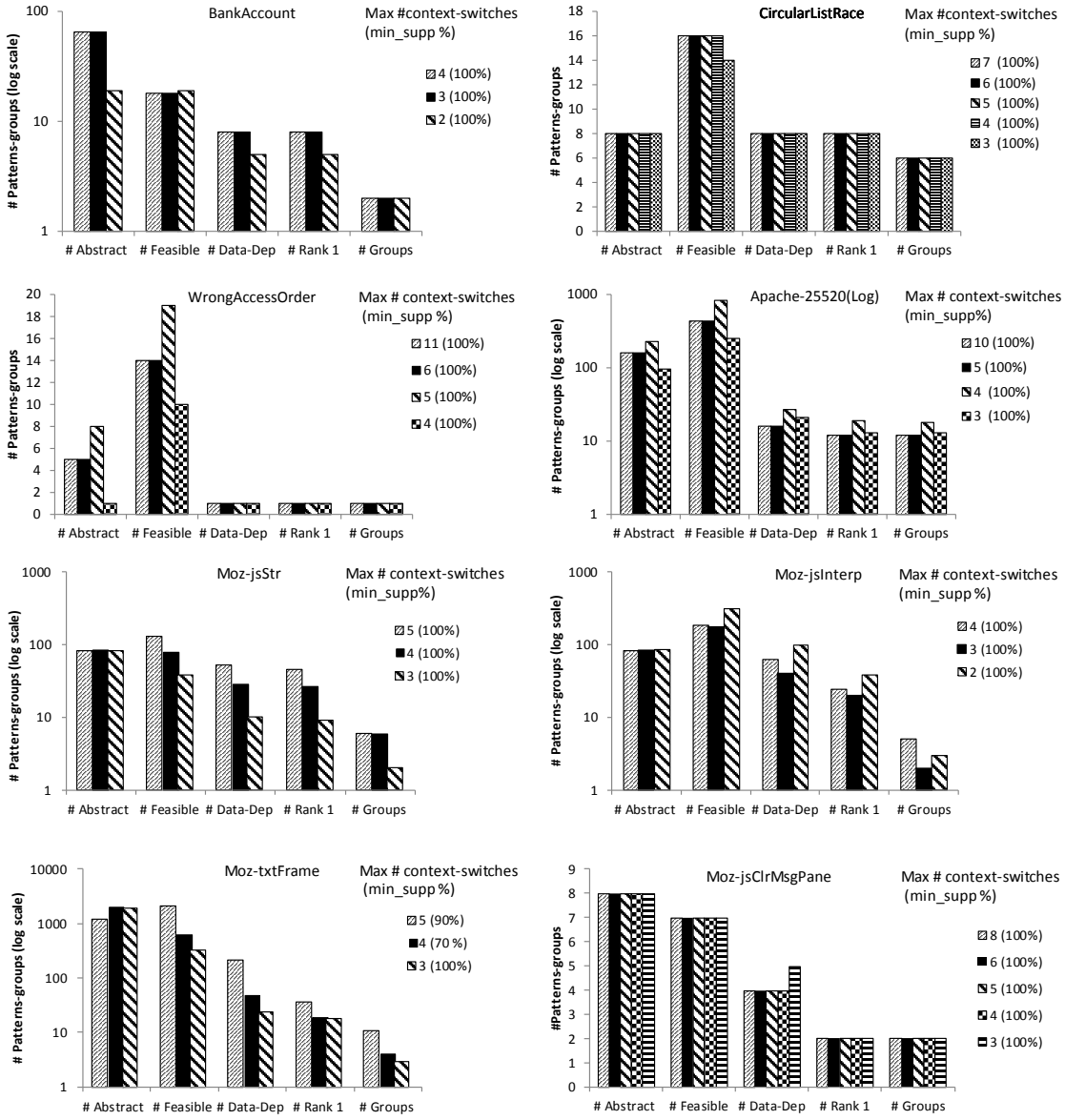


Figure 5.12: Mining results—context-switch bounded traces

In Figure 5.13, for every input dataset of Table 5.6 the patterns appeared at the top of the final result-sets are given. As we can see, corresponding to every case study the

patterns of different input datasets are similar in terms of the macros and the data-dependencies they contain. Consequently, all refer to the same concurrency bug. Due to the similarity between the patterns in Figure 5.13 and Figure 5.5, the explanations given in Section 5.4.2 for understanding bugs from patterns of Figure 5.5 are also applicable to the patterns of Figure 5.13. Only the pattern given for *Apache-25520(Log)* with *bound* = 3 is slightly different from other patterns of this case study, but reveals the same concurrency bug. In this pattern, the data-dependency between the events of the first macro reflects thread 1 appending an element to log. However, the data-dependency between first and second macros implies that the modification by thread 1 is not followed by a corresponding update of the log pointer, revealing an atomicity violation in accessing the log data-structure.

The experiments of this section show that even for input datasets containing a small number of traces (such as datasets with *bound* = 2 in BankAccount or *bound* = 3 in Apache-25520(Log)) the method is capable of generating useful bug explanation patterns.

5.4.6 Datasets with Randomly-chosen Traces

In Section 5.4.2, the failing and passing datasets for the two case studies *WrongAccessOrder* and *Apache-25520(Log)* contained the first 100 failing and 100 passing traces out of 1427 and 32930 traces available. In this section, we evaluate our method on the datasets generated by *randomly* choosing 100 failing and 100 passing traces. For each of these two case studies, we repeated the experiments 5 times, each time with different randomly generated failing and passing datasets. The results of applying Algorithm 5.2 on these datasets are given in Figure 5.14. As the diagrams show, we have a slight variation in the results of the algorithm for different random input datasets.

Figure 5.15 shows for both case studies the patterns ranked top in the final result-sets of the 5 different random datasets. The patterns are similar, hence revealing the same concurrency bug. The patterns for *Apache-25520(Log)* are similar to the pattern of the case study with *bound* = 3 in Figure 5.13. For *WrongAccessOrder*, the given patterns are similar to patterns of the case study in both Figures 5.13 and 5.5.

5.4.7 Threats to Validity

There is a limitation to the evaluation of our method. Although most of our case studies were used in other work, we have not applied our method to full large applications such as Mozilla and Apache. Since logging the traces and applying the abstraction offline may be impractical for these large applications, we plan to apply our abstraction technique online as the traces are being generated in future work.

5.5 Related Work

Given the ubiquity of multithreaded software, there is a vast amount of work on finding concurrency bugs. A comprehensive study of concurrency bugs [LPSZ08] identifies

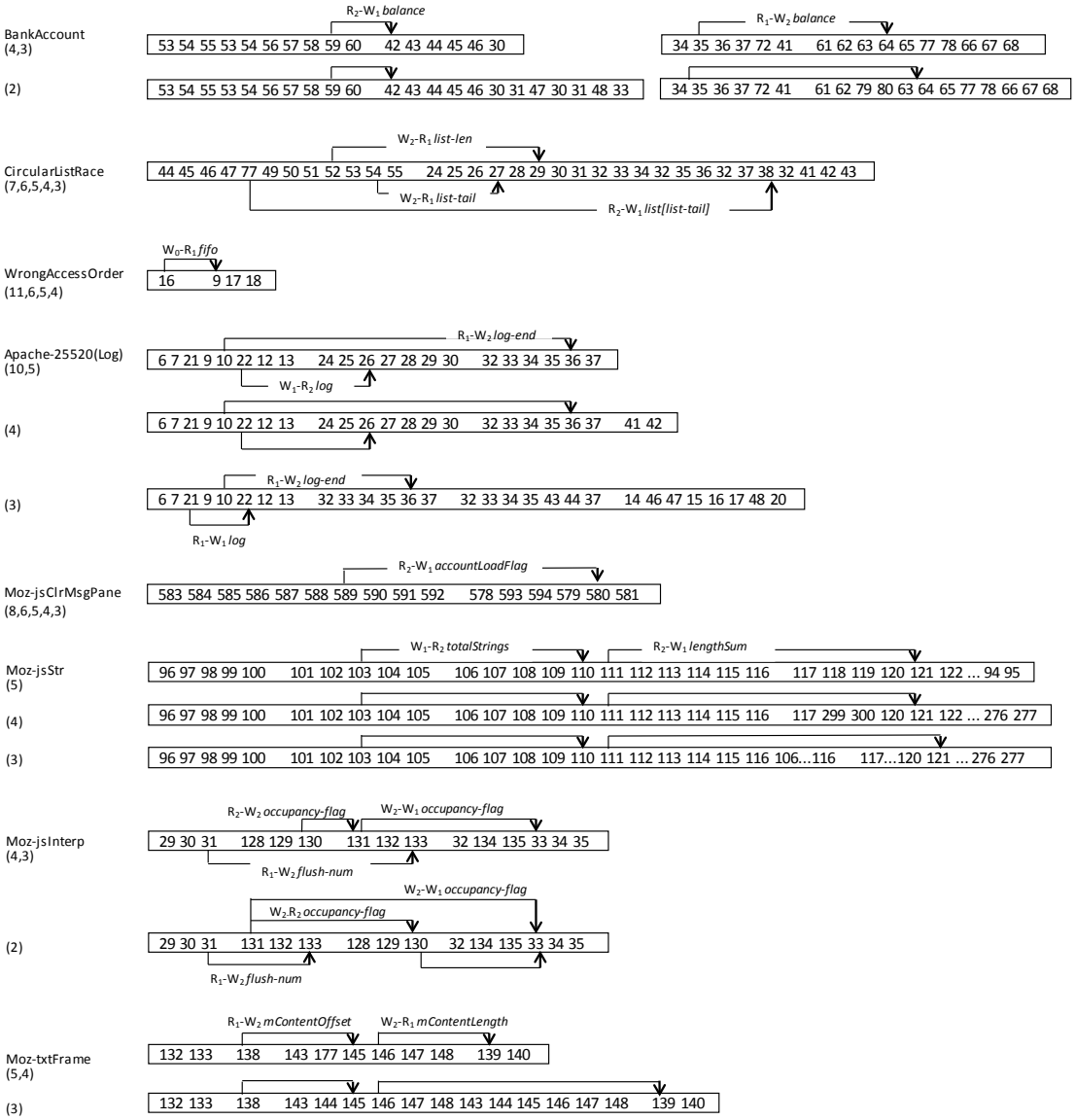


Figure 5.13: Bug explanation patterns—context-switch bounded traces (numbers in parenthesis shows the corresponding bounds used in generating the input datasets).

data races, atomicity violations, and ordering violations as the prevalent categories of non-deadlock concurrency bugs. Accordingly, most bug detection tools are tailored to identify concurrency bugs in one of these categories (See Chapter 2, Section 2.2). Avio [LTQZ06] detects single-variable atomicity violations by learning acceptable memory access patterns from a sequence of passing training executions, and then monitoring whether these patterns are violated. Svd [XBH05] is a tool that relies on heuristics to approximate atomic regions and uses deterministic replay to detect serializability violations. Lockset analysis [SBN+97a] and happens-before analysis [NM91b] are popular

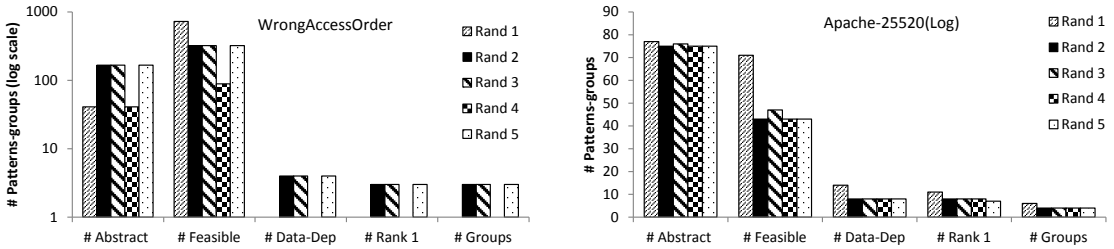


Figure 5.14: Mining results—randomly chosen traces

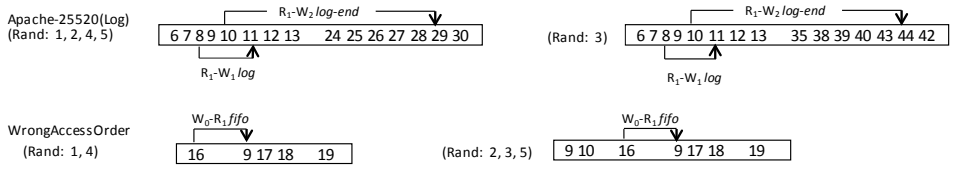


Figure 5.15: Bug explanation patterns—randomly chosen traces

approaches focusing only on data race detection. In contrast to these approaches, which rely on specific characteristics of concurrency bugs and lack generality, our bug patterns can reveal any type of concurrency bugs. The algorithms in [WS06] for atomicity violations detection rely on input from the user in order to determine atomic fragments of executions. Detection of atomic-set serializability violations by the dynamic analysis method in [HDVT08] depends on a set of given problematic data access templates. Unlike these approaches, our algorithm does not rely on any given templates or annotations. Bugaboo [LC09] constructs bounded-size context-aware communication graphs during an execution, which encode access ordering information including the context in which the accesses occurred. Bugaboo then ranks the recorded access patterns according to their frequency. Unlike our approach, which analyzes entire execution traces (at the cost of having to store and process them in full), context-aware communication graphs may miss bug patterns if the relevant ordering information is not encoded. Falcon [PVH10] and the follow-up work Unicorn [PVH12] can detect single- and multi-variable atomicity violations as well as order violations by monitoring pairs of memory accesses, which are then combined into problematic patterns. The suspiciousness of a pattern is computed by comparing the number of times the pattern appears in a set of failing traces and in a set of passing traces. Unicorn produces patterns based on pattern templates, while our approach does not rely on such templates. In addition, Unicorn restricts these patterns to windows of some specific length, which results in a local view of the traces. In contrast to Unicorn, we abstract the execution traces without losing information.

Methods of Chapters 3 and 4 have used pattern mining to explain concurrent counterexamples obtained by explicit-state model checking. In contrast to the method of this chapter, method of Chapter 3 mines frequent substrings instead of subsequences and method of Chapter 4 suggests a heuristic to partition the traces into shorter sub-traces. Unlike our abstraction-based technique, both of these approaches may result in the

loss of bug explanation sequences. Moreover, both methods are based on *contrasting* the frequent patterns of the failing and the passing datasets rather than ranking them according to their relative frequency. Therefore, their accuracy is contingent on the values for the *two* support thresholds of the failing as well as the passing datasets.

Statistical debugging techniques which are based on comparison of the characteristics of a number of failing and passing traces are broadly used for localizing faults in sequential program code. For example, a recent work [RFZO12] statically ranks the differences between a few number of similar failing and passing traces, producing a ranked list of facts which are strongly correlated with the failure. It then systematically generates more runs that can either further confirm or refute the relevance of a fact. In contrast to this approach, our goal is to identify problematic sequences of interleaving actions in concurrent systems.

Due to nondeterminism, *cyclic debugging* which is the most common methodology used for debugging sequential software can be ineffective for debugging concurrent programs [LMC87]. In cyclic debugging, when the programmer observes a failure, he postulates a set of underlying causes for the failure and accordingly inserts trace statements and breakpoints in the program code and reexecutes it. This methodology cannot be applied for debugging concurrent programs because successive executions of these programs do not necessarily produce the same results. Therefore, a number of techniques such as [LMC87] proposed for reproducing the execution behavior of concurrent programs. However, using the techniques such as [LMC87] only the execution behavior of a concurrent program can be reproduced for further analysis. The task of isolating and understanding the cause of failure still need to be done manually by the programmer. Our method differs from these methods as its goal is isolating the causes of failures automatically, hence, facilitating the task of debugging.

5.6 Summary

In this chapter, we introduced the notion of bug explanation patterns based on well-known ideas from concurrency theory, and argued their adequacy for understanding concurrency bugs. We explained how sequential pattern mining algorithms can be adapted to extract such patterns from logged execution traces. By applying a novel abstraction technique, we reduce the length of these traces to an extent that pattern mining becomes feasible. Our case studies demonstrate the effectiveness of our method for a number of synthetic as well as real world bugs.

5.7 Comparing the Proposed Mining Approaches: Chapters 3–5

To address the scalability issues in applying standard sequential pattern mining algorithms for explaining concurrency bugs, we proposed three approximation techniques in

Chapters 3–5. The substring and subtrace mining methods of Chapters 3 and 4 have been developed by exploiting the characteristics of the traces in messages passing concurrent systems. Since substrings of some specific length ℓ can be extracted from a dataset Σ containing m sequences of maximum length n in worst case in $O(mn)$, the substring mining method runs in polynomial time. In practice, we have shown that the average running time of the method is 52.44 sec. for datasets with 16697 average sequence length, 134629 maximum sequence length and 9028 average number of sequences (Section 3.3). In substring mining method, we abandon the feature of arbitrary distance between the events of a subsequence which we consider it essential for understanding problematic context switches. Therefore, the extracted substrings can't reveal explanatory sequences as defined in Section 3.1.2, but only portions of these sequences. However, we have shown the effectiveness of these small portions in facilitating the search for explanatory sequences by computing a quantitative score (Table 3.2).

We improved our bug explanation technique in Chapter 4 by proposing a method for extracting sequences of events which do not necessarily occur contiguously inside traces. In other words, the method of Chapter 4 mines subsequences from traces as opposed to substring mining method of Chapter 3. In order to make the subsequence mining problem more tractable, the approximation technique of Chapter 4 reduces the length of the traces by partitioning them into subtraces. In practice, we have shown that the partitioning technique can reduce the length of the traces on the average by 96.5% resulting into subtraces with average length of 24. However, the partitioning technique exploits the characteristics of traces of non-terminating communication protocols. Therefore, the applicability of subtrace mining method is limited to traces which can be partitioned into subtraces with minimum dependencies across the subtraces. The dependency across the subtraces refer to, for instance, data dependencies between the events in different subtraces or the dependency between a send event and its corresponding receive event. Moreover, since subtraces provide a local view of the traces, as we discussed of "Threats to Validity" in Section 4.2.2, there may exist concurrency bugs for which the method fails to provide an explanation.

The abstraction method of this chapter (Chapter 5) addresses the main shortcomings of the two previous methods. In contrast to substring mining method of Chapter 3, it mines explanatory sequences which can occur as subsequences of traces. To improve scalability, it reduces the length and the number of distinct events in traces by using an abstraction technique. Abstract traces as opposed to subtraces preserve the ordering information between all the events of original traces including the context switches between the threads. We have shown in practice that the length of the traces is reduced on average by 91% using the abstraction technique. Therefore, the pattern mining from the traces up to 13,000 events took only 87ms on average. Moreover, we proved that the abstraction preserves the original patterns. However, it may introduce spurious patterns which can be filtered efficiently after mining abstract patterns. Although our abstraction technique has been shown to be efficient in reducing the length of the traces and making the pattern mining problem tractable, it relies on the availability of a dataset of logged traces which

can make it impractical for large applications such as Mozilla or Apache.

CHAPTER 6

Explaining Concurrency Bugs using Interpolant-based Slicing

In Chapters 3-5, we proposed *anomaly detection* techniques based on mining datasets of concurrent traces for explaining concurrency bugs. In this chapter, we propose a different approach for concurrency bug explanation. This approach performs *slicing* of a single failing concurrent trace using a *proof-based* technique. In Chapter 5, we have shown the efficiency and effectiveness of the abstraction based mining method in explaining concurrency bugs. However, this method relies on the availability of a dataset of logged traces. The quality of the output of the method can also be dependent on the given failing and passing datasets even though the experimental results in Section 5.4 show zero false positives for the given case studies.

The underlying principles of the interpolation-based slicing method of this chapter allows us to provide a soundness proof for the method. Moreover, for constructing slices the method analyzes only one single failing trace.

As we have seen in Chapter 2, dynamic slicing techniques [Tip95] were introduced to effectively narrow down the search for the root cause of the failure in a failing trace by automatically removing irrelevant statements from the trace. In these techniques, data and control dependencies are taken into account in order to remove the statements which do not impact the failing state via any chain of dependencies. However, the main limitation of the dynamic slicing techniques is that they do not consider the *semantics* of a failure, which can result in irrelevant statements being retained in the slice. For example, in the failing trace given in Figure 6.1, statement 2 is irrelevant to the assertion violation. However, dynamic slicing which considers dependencies at the syntactic level cannot exclude this statement. Recent work [ESW12, CESW13, MSTC14] showed how *interpolation* can be used to construct semantics-aware slices in sequential software. Here, a failing trace is translated into an unsatisfiable logical formula. From a proof of

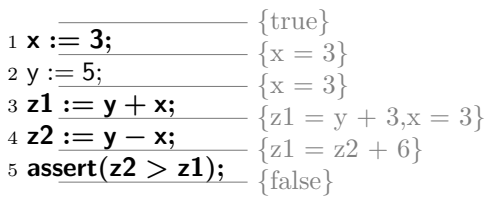


Figure 6.1: A trace annotated with interpolants explaining the cause of an assertion violation. [MSTC14]

unsatisfiability interpolants are extracted that capture the reason why the trace failed. These interpolants are then used to construct a slice of the failing trace that abstracts from the irrelevant statements and explains the faulty behavior. This approach produces a slice of the original trace annotated with assertions (the obtained interpolants) showing the relevant values and variables to the failure. The bold statements in Figure 6.1 constitute a sample slice with assertions produced by this interpolation-based slicing technique. Notice that the outcome of assertion at line 5 is semantically independent of y, so the statement at line 2 can be sliced.

In this chapter, we lift the existing interpolation-based slicing techniques to a concurrency setting. To this end, we take into account control- and data-dependency between threads in addition to intra-thread dependencies. We stress that considering inter-thread data-dependencies allows us to isolate hazards such as race conditions and atomicity violations, which constitute the predominant class of non-deadlock concurrency bugs [LPSZ08]. Therefore, the approach of this chapter do not rely on characteristics specific to each type of concurrency bug, hence providing a general framework for concurrency bug explanation.

We have implemented our approach in an automated debugging tool and applied it to failing traces generated from concurrent C programs using the directed testing tool ConCrest [FHRV13]. We present two detailed case studies that demonstrate the effectiveness of hazard-aware interpolation for explaining common types of concurrency bugs. Moreover, we report on an evaluation of our approach on a benchmark suite taken from the literature that contain bugs found in real-world software such as Apache, MySQL, and GCC [KKW15]. We observed that our new analysis yields precise explanations of concurrency bugs. Often these explanations are as concise as the explanations that a human would generate manually. We further found that, on average, our generated slices yield a significant reduction of the number of variables and the length of the considered traces.

6.1 Preliminaries

In this section, we formalize concurrent executions, traces, and symbolic traces. We introduce the notion of interpolant and explain how it is used for slicing traces of

sequential software.

6.1.1 Concurrent Programs

Similar to the method of Chapter 5, we consider shared-memory concurrent programs composed of k *threads* with indices $\{1, \ldots, k\}$ and a finite set $\mathbb{G}$ of *shared variables*. (This paragraph and the following can be skipped since they are similar to the content of Section 5.1.1.) Each thread T_i where $1 \leq i \leq k$ has a finite set of *local variables* $\mathbb{L}_i$. The set of all variables is then defined by $\mathbb{V} \stackrel{\text{def}}{=} \mathbb{G} \cup \bigcup_i \mathbb{L}_i$, where $1 \leq i \leq k$. Each thread is represented by a control flow automaton (CFA) $\langle L_T, E_T, \lambda_T, L_{0_T} \rangle$, where L_T and E_T are sets of nodes and edges, respectively, L_{0_T} the initial node, and λ_T is a labeling function linking edges to *atomic instructions*. The set of nodes L_T corresponds to program locations of the thread, and the set of edges E_T defines the flow of control. We use guarded statements to represent atomic instructions. Let $\mathbb{V}_i = \mathbb{G} \cup \mathbb{L}_i$ (for $1 \leq i \leq k$) denote the set of variables accessible in thread T_i. An atomic instruction from thread T_i is either a guarded statement $\mathsf{assume}(\varphi) \triangleright \tau$ or an assertion $\mathsf{assert}(\varphi)$ where φ is a predicate over $\mathbb{V}_i$ and τ is an assignment of the form $v := \phi$ (where $v \in \mathbb{V}_i$ and ϕ is an expression over $\mathbb{V}_i$). The predicate φ must be true for the assignment τ to be enabled. It must be also true when $\mathsf{assert}(\varphi)$ is executed, otherwise a failure occurs.

The guarded statement has the following three variants: (1) when the guard $\varphi = \mathsf{true}$, it can model ordinary assignments in a basic block, (2) when the assignment τ is empty, the conditions $\mathsf{assume}(\varphi)$ and $\mathsf{assume}(\neg\varphi)$ can model the execution of a branching statement (conditional statement) $\mathsf{if}(\varphi) - \mathsf{else}$, and (3) with both the guard and the assignment, it can model an atomic *check-and-set* operation, which is the foundation of all types of concurrency primitives [HS08]. For example, acquiring and releasing a lock l in a thread with index i is modeled as $\mathsf{assume}(l = 0) \triangleright l := i$ and $\mathsf{assume}(l = i) \triangleright l := 0$, respectively. Fork and join can be modeled in a similar manner using auxiliary synchronization variables. In the following, assumptions $\mathsf{assume}(\varphi)$ are also denoted by $[\varphi]$. Figure 6.2 shows an example of a CFA.

As we have seen in Chapter 2 (§2.1.1), statement B is *control dependent* on statement A, if outcome of A determines whether B is executed or not. We define *scopes* for conditional statements according to the notion of control dependency:

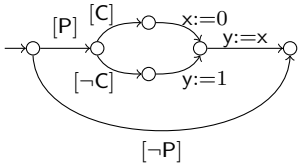


Figure 6.2: CFA

Definition 18 (Scope for Conditions)**.** *An atomic instruction s is in the scope of a conditional statement c such as $\mathsf{if}(\varphi)$ which is modeled as $\mathsf{assume}(\varphi)$, if s is control-dependent on c or in the scope of a condition that is control-dependent on c.*

In Figure 6.2, $\mathsf{x} := 0$ is in scope of condition C, whereas $\mathsf{y}{:=}\mathsf{x}$ is not. Both assignments, however, are in scope of condition P, since they are unreachable if P does not hold.

Executions. An *execution* $\rho = S_0, e_1, S_1, ..., S_{n-1}, e_n, S_n$ is an alternating sequence of states S_i and events e_i corresponding to some interleaving of atomic instructions from the threads of the program (cf. Definition 1). Each state S is a valuation of the variables $\mathbb{V}$. Each event e corresponds to the execution of an atomic instruction of a thread. For each i, the execution of e_i in state S_{i-1} leads to state S_i. We call the execution $\rho = S_0, e_1, S_1, ..., S_{n-1}, e_n, S_n$ *passing* if for each assertion $e_i = \mathsf{assert}(\varphi)$, the predicate φ evaluates to true at state S_{i-1}; otherwise we call the execution *failing*. Consequently, in a failing execution the last event e_n corresponds to a failing assertion.

Traces. A *trace* $\sigma = e_1, ..., e_n$ is a finite sequence of events that corresponds to some interleaving of atomic instructions from the threads of the program. A trace σ can correspond to several executions, however it is considered as *feasible* if it has at least one corresponding execution. A feasible trace is referred to as a *passing trace* if all the corresponding executions are passing. If a feasible trace has at least one corresponding failing execution, it is referred to as a *failing trace*. Intuitively, a failing trace results from the execution of a failing test case that does not satisfy some specification ψ. The specification ψ is modeled as an assertion $assert(\varphi)$ which corresponds to the last event e_n in the failing trace.

Given a trace $\sigma = e_1, ..., e_n$, we use $\sigma[i, j]$ to denote the *subtrace* $e_i, e_{i+1}, \ldots, e_{j-1}, e_j$ of σ including the events e_i and e_j and $\sigma(i, j)$ to denote the subtrace $e_{i+1}, \ldots, e_{j-1}$ excluding the events e_i and e_j. Moreover, we use $\sigma[i]$ to refer to the i^{th} event of σ which is equal to e_i.

Symbolic Traces. We use first-order logic (defined as usual) with background theories commonly used in software verification (such as arithmetic, bit-vectors, arrays and uninterpreted functions) to represent program expressions and predicates. true (false) represents the predicate that is always true (false).

We encode a trace $\sigma = e_1, ..., e_n$ as a quantifier free first-order logic formula $\Phi(\sigma)$, which is achieved by transforming the trace into Static Single Assignment (SSA) form [CFR+91]. We refer to $\Phi(\sigma)$ as a *trace formula*. In SSA form, each variable is defined exactly once. Here, an event *defines* a variable v if v appears in the left-hand side of an assignment; an event uses v if v appears in a condition (an assume or an assert) or the right-hand side of an assignment. The standard mechanism to transform a trace into SSA form is to subscript each definition of a variable with a unique version number; consequently, each definition is uniquely identified by the corresponding SSA variable. In general SSA representation, ϕ-functions resolve conflicting definitions at a control-flow merge point. Algorithms to convert a program into SSA form are described in [CFR+91] and [Muc97, §8.11]. Since we restrict ourselves to traces in which each thread has a single control path, ϕ-functions are not required. Therefore, for traces (without control-flow merge points) for transforming to SSA form it suffices to increase the version number of a variable each time it is assigned and refer to the latest version of each variable in conditions and right-hand sides of assignments. For example, for the trace $\sigma \stackrel{\text{def}}{=} [\mathsf{x} < 1], \mathsf{x} := \mathsf{y}$, the SSA

form is: $\mathsf{ssa}(\sigma) = [\mathsf{x}_0 < 1]\,\mathsf{x}_1 := \mathsf{y}_0 + 1$. In the following, we use $\mathsf{ssa}(\sigma)[i]$ to denote the i^{th} element of $\mathsf{ssa}(\sigma)$ which corresponds to the i^{th} event of σ in the SSA form.

To construct the trace formula $\Phi(\sigma)$, we encode the individual events in $\mathsf{ssa}(\sigma)$ into logical formulas as follows:

$$\Phi(\sigma[i]) \stackrel{\text{def}}{=} \begin{cases} (v_k = \phi) & \text{if } \mathsf{ssa}(\sigma)[i] \text{ is } v_k := \phi \\ (\varphi) & \text{if } \mathsf{ssa}(\sigma)[i] \text{ is } \mathsf{assume}(\varphi) \\ ((\varphi) \wedge (v_k = \phi)) & \text{if } \mathsf{ssa}(\sigma)[i] \text{ is } \mathsf{assume}(\varphi) \triangleright v_k := \phi \\ (\varphi) & \text{if } \mathsf{ssa}(\sigma)[i] \text{ is } \mathsf{assert}(\varphi) \end{cases} \tag{6.1}$$

where $1 \leq i \leq |\sigma|$. The trace formula $\Phi(\sigma)$ is then a sequence of conjuncts each of which corresponds to encoding of individual events in $\mathsf{ssa}(\sigma)$ according to (6.1):

$$\Phi(\sigma) \stackrel{\text{def}}{=} \Phi(\sigma[1]) \wedge \Phi(\sigma[2]) \ldots \wedge \Phi(\sigma[n]).$$

For example, for the trace $\sigma \stackrel{\text{def}}{=} [\mathsf{x} < 1]; \mathsf{x} := \mathsf{y}$, we obtain: $\Phi(\sigma) = (\mathsf{x}_0 < 1) \wedge (\mathsf{x}_1 = \mathsf{y}_0 + 1)$. Any satisfying assignment to $\Phi(\sigma)$ corresponds to an execution (passing) of σ; note that if all variables in $\Phi(\sigma)$ are initialized before being used, $\Phi(\sigma)$ has only one unique satisfying assignment.

6.1.2 Interpolation-based Slicing for Sequential Software

In this section, we recall the interpolation-based slicing approach presented in [ESW12, CESW13, MSTC14]. In this approach *interpolants* represent over-approximation of the set of reachable states during an execution and are used for slicing a failing trace.

Definition 19 (Interpolant)**.** *Let A and be B be a pair of first-order formulas such that $A \wedge B$ is unsatisfiable. An* interpolant *of A and B is a first-order formula I such that $A \Rightarrow I$, $B \Rightarrow \neg I$, and* $\mathrm{Var}(I) \subseteq \mathrm{Var}(A) \cap \mathrm{Var}(B)$*. (*Var *denotes the set of free variables in a formula.)*

Definition 19 corresponds to the definition of interpolants in [McM05] under the assumption that all non-logical symbols in A and B are interpreted. Similarly, we can define an *interpolation sequence* for a sequence of formulas:

Definition 20 (Interpolation Sequence)**.** *Let $A_1, \ldots, A_n$ be a sequence of first-order formulas whose conjunction is unsatisfiable. Then $I_0, \ldots, I_n$ is an* interpolation sequence if:

- $I_0 = \mathsf{true}$ and $I_n = \mathsf{false}$,
- for all $1 \leq i < n$: $A_1 \wedge A_2 ... \wedge A_i \Rightarrow I_i$, $A_{i+1} \wedge A_{i+2} ... \wedge A_n \Rightarrow \neg I_i$
- for all $1 \leq i < n$: $\mathrm{Var}(I_i) \in (\mathrm{Var}(A_1 \wedge \ldots \wedge A_i) \cap \mathrm{Var}(A_{i+1} \wedge \ldots \wedge A_n))$.

In order to slice a failing trace using interpolants, according to Definitions 19 and 20, we need to construct an unsatisfiable formula for computing interpolants. Given a failing

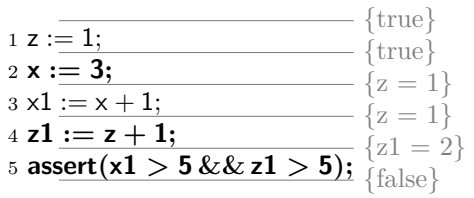


Figure 6.3: Unsound slice. [MSTC14]

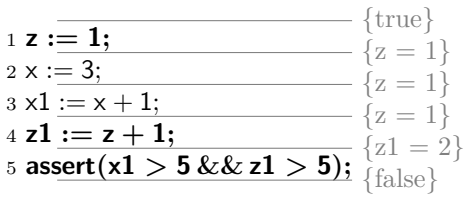


Figure 6.4: Sound slice. [MSTC14]

trace $\sigma = e_1, ..., e_n$, in which all the variables are *initialized* according to a failing test case, the last event e_n corresponds to a failing assertion like $\mathsf{assert}(\varphi)$. Since φ evaluates to false in σ, the trace formula $\Phi(\sigma)$ is an unsatisfiable formula. Using Definition 20, we can then compute interpolants for each position $i, 1 \leq i < n$ in σ. Each interpolant $I_i, 1 \leq i < n$ represents an over-approximation of the states reachable at position i in σ such that the execution of events $e_{i+1}, ..., e_n$ from a state in I_i still results in the assertion violation. Ermis et al. [ESW12] and Christ et al. [CESW13] refer to interpolants I_i computed for each position $i, 1 \leq i < n$, in σ as *error invariants*.

In [ESW12, CESW13], an error invariant I is considered *recurring* for positions $i \leq j$ if I is an error invariant for both i and j. According to [ESW12, CESW13], statements between recurring error invariants are "not needed to reproduce the failure." Therefore, they can be sliced from the failing trace. Moreover, the interpolants generated for slicing irrelevant statements serve as an annotation characterizing at each point in the slice the states eventually leading to a failing assertion. For example, in Figure 6.1 the error invariant $\{x = 3\}$ is recurring for positions 1 and 2, therefore statement 2 is sliced away since it does not have an effect on assertion violation at line 5. Moreover, in this figure, error invariants revealing the variables and values which are relevant to the assertion violation annotate the statements in the slice.

Interpolants I_i are not unique, therefore a failing trace can be sliced in different ways using interpolants. However, we are looking for *sound* slices in the sense of Definition 21 below:

Definition 21 (Sound Slice)**.** *A slice of a failing trace $\sigma = e_1, ..., e_n$ is a trace $\eta = e'_1, ..., e'_m$ such that $e'_1 = e_{i_1}$, $e'_2 = e_{i_2}$, ..., $e'_m = e_{i_m}$ where $1 \leq i_1 < i_2 < \ldots < i_m \leq n$. The slice η is* sound *if the formula $\Phi(\eta)$ is unsatisfiable.*

For example, in Figure 6.3 the computed interpolation sequence results in an unsound slice, although interpolants are valid according to Definition 20. Based on the computed interpolation sequence, statements 1 and 3 are irrelevant to the assertion violation since they are surrounded by recurrent error invariants, $\{\text{true}\}$ and $\{z = 1\}$, respectively. However, by removing these two statements from the failing trace, variables z and $x1$ become unconstrained which makes the formula corresponding to the resulting slice satisfiable.

```
procedure SLICE(η,(I_0, ..., I_n))
    if ∃i, j . I_i is recurring for j > i then
        η ← (e_1, ..., e_i, e_{j+1}, ..., e_n)
        SLICE(η, (I_0, ..., I_i, I_{j+1}, ..., I_n))
    else
        return η
    end if
end procedure
```

Figure 6.5: Interpolation-based Slicing Algorithm

Murali et al. [MSTC14] point out that in order to obtain a sound slice by removing statements between recurring error invariants we need to compute an *inductive interpolant sequence*:

Definition 22 (Inductive Interpolant Sequence)**.** *Let $A_1, \ldots, A_n$ be a sequence of first-order formulas whose conjunction is unsatisfiable. The interpolation sequence $I_0, I_1, \ldots, I_n$ in Definition 20 is* inductive *if for all I_i, $1 \leq i \leq n$, we have $I_{i-1} \wedge A_i \Rightarrow I_i$.*

Moreover, the inductive interpolant sequence for the failing trace σ corresponds to a Hoare proof of $\{\mathsf{true}\}\ \sigma\ \{\mathsf{false}\}$ [McM06, MSTC14]. ($\{I_{i-1}\}\ \sigma[i]\ \{I_i\}$ is a valid Hoare triple for each $1 \leq i \leq n$.) It is easy to see that the interpolant sequence in Figure 6.3 is not inductive, because $(\mathsf{true}) \wedge \mathsf{x} = 3 \not\Rightarrow (z = 1)$. A possible inductive interpolant sequence for the trace in Figure 6.3 is given in Figure 6.4. In this figure, the resulting slice is sound according to Definition 21. Given an inductive interpolant sequence and a failing trace, Figure 6.5 shows the algorithm for interpolation-based slicing. Initially the parameter η in this algorithm is equal to the original trace σ. According to [MSTC14, Theorem 1], if $(I_0, \ldots, I_n)$ in Figure 6.5 is an inductive interpolant sequence, the resulting slice will be sound (Definition 21).

Control-flow Sensitive Slicing

The encoding of the failing trace using the trace formula $\Phi(\sigma)$ defined above fails to capture control dependencies. Therefore, if there exist conditional statements relevant to the failure, they will not be included in the computed slice. In Figure 6.6, according to the computed inductive interpolant sequence the branch at line 3 is sliced away although it is relevant to the failure. Therefore, the resulting slice does not reflect the fact that the branch of the conditional statement has to be taken for the failure to occur. The encoding presented in [CESW13] lifts this restriction by prefixing the conjuncts encoding assignments in $\Phi(\sigma)$ with conditions of the respective scopes. Algorithm 6.1 constructs a control-flow sensitive encoding $\Phi_{\mathsf{fs}}(\sigma)$ of a failing trace [CESW13]. In this algorithm, the conjunction of the conditions of the respective scopes is referred to as a *guard*.

An inductive error invariant for the encoding $\Phi_{\mathsf{fs}}(\sigma)$ constructed by Algorithm 6.1 induces a control-flow sensitive slice (cf. Definition 4 and Theorem 6 in [CESW13]):

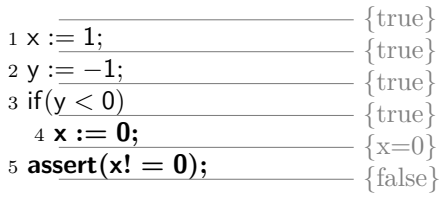


Figure 6.6: A flow-insensitive slice not containing the relevant branch.

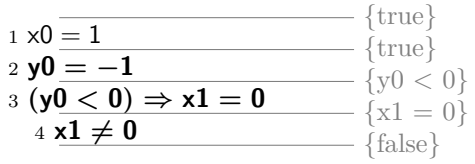


Figure 6.7: Control-flow sensitive slice of the trace in Figure 6.6 which contains the relevant branch.

Definition 23 (Control-flow sensitive Slice)**.** *Let σ be a failing trace. A (sound) slice η is* control-flow sensitive *if for every event $\eta[k] = \sigma[i]$ and every assumption $\sigma[j] = [C]$ such that $\eta[k]$ is in scope of $\sigma[j]$, there is some prefix $\eta[1,h]$ of $\eta[1,k]$ (with $h < k$ such that $\eta[h]$ precedes and $\eta[h+1]$ succeeds or equals $\sigma[j]$ in σ) such that $\eta[1,h]; \texttt{assert}(\neg C)$ is a failing trace.*

Intuitively, the definition requires that η justifies that every branch containing a relevant statement will be taken.

Theorem 6.1.1. *Let σ be a failing trace and let $I_0, I_1, \ldots, I_{(|\sigma|-1)}, I_{|\sigma|}$ be error invariants (with $I_0 = \mathsf{true}$ and $I_{|\sigma|} = \mathsf{false}$) obtained from an inductive interpolant sequence for $\Phi_{\mathsf{fs}}(\sigma)$. Let η be the slice obtained from σ by removing each subtrace $\sigma[i,j]$ for which I_{i-1} is recurrent. Then η is a sound control-flow sensitive slice for σ.*

Proof (sketch). Since $I_0, I_1, \ldots, I_{(|\sigma|-1)}, I_{|\sigma|}$ is an inductive interpolant sequence, according to [MSTC14, Theorem 1] the induced slice using this inductive sequence is sound. It remains to show that η is control-flow sensitive. Assume that $\eta[j] = \sigma[i]$ (i.e., $\sigma[i]$ is not sliced), and that C encodes the conditions of its scope (the guard as in Algorithm 6.1). Therefore, we have $I_{i-1} \Rightarrow C$ because otherwise I_{i-1} is also an error invariant for the subsequent position, and $\sigma[i]$ should have been sliced. Moreover, due to the soundness of η, I_{i-1} is established by a prefix of $\eta[1,j]$.

□

The control-flow sensitive encoding of the trace in Figure 6.6 ($\Phi_{\mathsf{fs}}(\sigma)$) along with the computed interpolant sequence are given in Figure 6.7. As we can see, the resulting slice shows that the initialization of variable y at statement 2 as well as the condition at statement 3 are relevant for the assertion violation.

6.2 Interpolation-based Slicing for Concurrent Traces

In the following, we adapt the interpolation-based slicing technique discussed in Section 6.1.2 to concurrent traces. Although the interpolation-based slicing technique for sequential software (introduced in Section 6.1.2) is readily applicable to concurrent

Algorithm 6.1 Encoding control-flow sensitive trace formulas [CESW13]

Input: Failing trace σ of length $|\sigma|$
Output: Control-flow sensitive trace formula $\Phi_{\mathsf{fs}}(\sigma)$

```
1: for i=1 to |σ| do
2:     if σ[i] is assignment then
3:         let guard ≝ ⋀{Φ(σ)[j] | σ[i] is in scope of σ[j]} in
4:         Φ_fs(σ)[i] := (guard ⇒ Φ(σ)[i]) ;
5:     else
6:         Φ_fs(σ)[i] := true
7:     end if
8: end for
```

traces, a straight-forward application thereof, however, ignores several characteristics of concurrent traces (in particular inter-thread data dependencies), resulting in slices that may not reflect the underlying problem. We illustrate the inadequacy of the technique discussed in Section 6.1.2 in slicing concurrent traces using a well-understood example of an atomicity violation.

6.2.1 Motivating Example

Atomicity violations are one of the most common types of non-deadlock concurrency bugs [LPSZ08]. Figure 6.8 shows a classic atomicity violation, in which two code fragments non-atomically update the balance of a bank account (stored in the shared variable balance) at locations L_2 and L_2' (similar to the motivating example in Chapter 5, Section 5.1.3). The example does not contain a data race, since balance is protected by the lock ℓ. The array t_array contains the sequence of amounts to be transferred, partitioned into deposits and withdrawals. The threads T_1 and T_2 both execute concurrently, depositing and withdrawing amounts of money, respectively. Figure 6.9 shows a failing execution of the implementation. In this execution, in a loop Thread T_1 executes three deposits and Thread T_2 executes two withdrawals. First, Thread T_2 stores the value of the current balance in a thread-local variable bal. At this point, Thread T_1 interferes and updates the value of balance by performing three deposit transactions. Thread T_2, then, proceeds with the now stale value stored in bal and stores the result of the two withdrawal transactions in balance. Consequently, the execution results in a discrepancy of the expected and the actual balance on the account. Applying the technique in Section 6.1.2 on the concurrent trace in Figure 6.9 results in a slice given in Figure 6.10 (annotations are not shown in this figure). As it can be seen in Figure 6.10, the deposits executed by Thread T_1 have been sliced away. This is because Thread T_2 subsequently updates the shared variable balance, ensuring that the values written by T_1 do not affect the assertion. Therefore, the slice does not reflect the problematic interference of Thread T_1 which results in the inconsistent value of balance. To remedy this problem, we take into account the inter-thread data dependencies as well as control dependencies via locking mechanism.

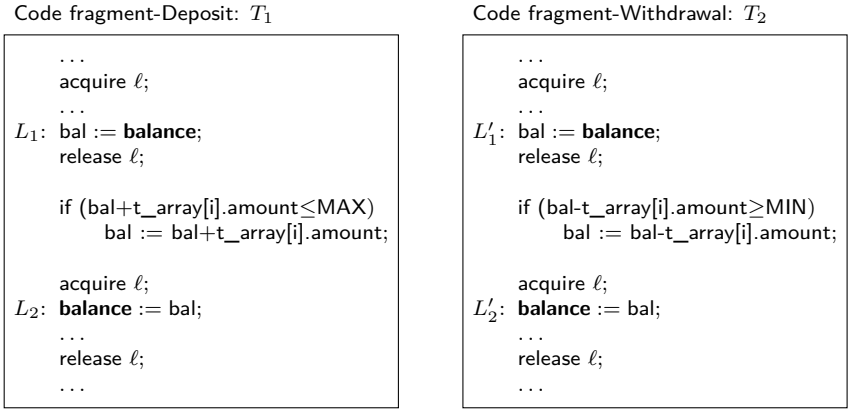


Figure 6.8: Non-atomic update of bank account balance

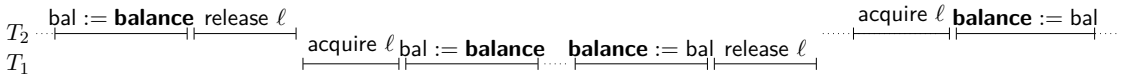


Figure 6.9: Fragment of a Faulty Program Execution for Example in Figure 6.8

T_0 : t_array[i].amount := 20//1st withdrawal amount
...

T_2 : bal := **balance**;
T_2 : release ℓ;
...
T_2 : bal := bal − t_array[i].amount
T_2 : acquire ℓ;
T_2 : **balance** := bal;
...

Figure 6.10: Fragment of the computed slice for the trace in Figure 6.9 using the techniques in § 6.1.2

6.2.2 Inter-thread Control Dependency

In Section 6.1.1, we modeled locks (as well as other synchronization mechanisms) using guarded assignments.

$$\begin{aligned} \mathsf{acquire}\ \ell &\stackrel{\text{def}}{=} \mathsf{assume}(\ell = 0) \triangleright \ell := \mathsf{tid} \\ \mathsf{release}\ \ell &\stackrel{\text{def}}{=} \mathsf{assume}(\ell = \mathsf{tid}) \triangleright \ell := 0 \end{aligned} \tag{6.2}$$

Here, a value of 0 indicates that the lock ℓ is available, and the non-zero value tid represents the unique identifier of the current thread. By virtue of acquire and release being modeled using atomic guarded commands (Equation 6.2), the notion of a scope

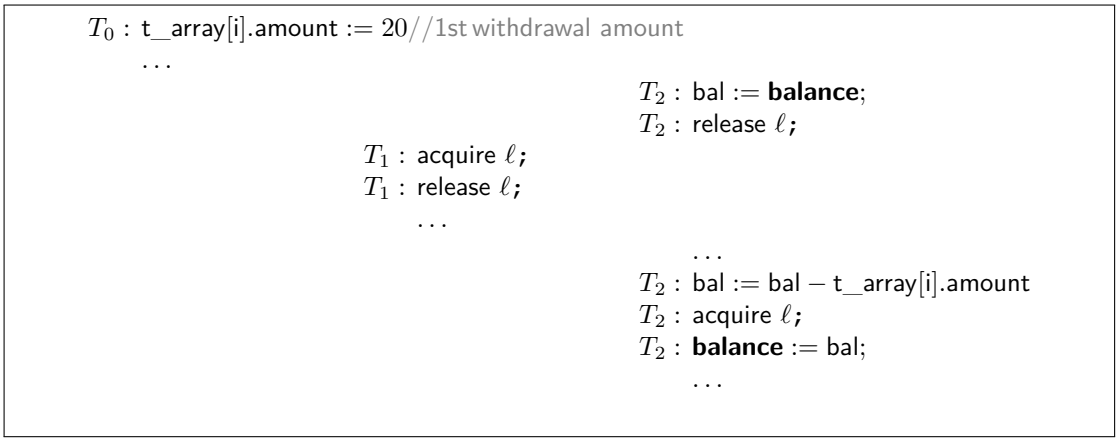


Figure 6.11: Fragment of a control-flow sensitive slice for the concurrent trace in Figure 6.9

also extends to locks. However, we use a *dynamic* scope for locks, which we deem more intuitive:

Definition 24 (Lock Scope)**.** *An event $\sigma[k]$ issued by a thread t is* in scope of a lock ℓ *in trace σ if there exists an event $\sigma[i] = \mathsf{acquire}(\ell)$ (where $i < k$) issued by thread t such that for all j with $i < j < k$ we have that $\sigma[j] \neq \mathsf{release}(\ell)$ if $\sigma[j]$ is an event issued by thread t.*

In Algorithm 6.1 at Line 3 locking scopes according to Definition 24 need also to be taken into account. Therefore, the modified encoding $\Phi_{\mathsf{fs}}(\sigma)$ which contains guards related to locking scopes can be used for generating control-flow sensitive slices for concurrent traces with locks. For example, the modified encoding results in the slice given in Figure 6.11 for the failing trace of bank account case study in Figure 6.9.

As it can be seen in Figure 6.11, by incorporating inter-thread control dependencies via locking mechanism the resulting slice has been improved. While the slice in Figure 6.10 has ignored Thread T_1 altogether, the slice in Figure 6.11 contains acquiring and releasing lock statements of Thread T_1. From this slice, it can be understood that there was a context-switch between reading and writing of the shared variable balance in Thread T_2. Therefore, it can be inferred that perhaps this context-switch caused the inconsistency in the value of balance. However, it is not explicit in the slice the updates of balance in Thread T_1 which caused the atomicity violation. This is due to the fact that we do not consider inter-thread data dependencies in the encoding of concurrent traces.

6.2.3 Inter-thread Data Dependency

A data-dependence is a constraint arising from the flow of data between statements. We use the terminology of [Muc97]:

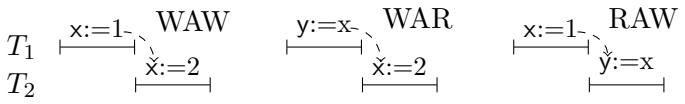


Figure 6.12: Inter-thread Data-dependencies

Read-after-write (RAW) If statement A writes a value that is read by statement B, then the two statements are *flow-dependent*.

Write-after-read (WAR) An *anti-dependence* occurs when statement A reads a value that is later updated (over-written) by B.

Write-after-write (WAW) An *output-dependence* exists if A as well as B set the value of the same variable.

While this definition also applies to single threads, we concern ourselves exclusively with *inter-thread* data-dependencies. Figure 6.12 illustrates data-dependencies between the threads. In a trace σ, a data-dependency between different threads can indicate a conflicting access due to a *data race*, an *atomicity violation*, or an *order violation* which we refer to all as *hazards*.

Flow-dependence is taken into account by the vanilla SSA encoding $\Phi(\sigma)$ (Section 6.1.1) since the SSA form represents use-definition pairs, and therefore also flow-dependence explicitly. However, anti- and output-dependencies are not explicit in the vanilla SSA encoding $\Phi(\sigma)$ used in Sections 6.1.2 and 6.2.2.

Similar to control-flow merge points in sequential programs, inter-thread dependencies in a concurrent program give rise to conflicting definitions of shared variables. The Concurrent SSA (CSSA) form of traces presented in [WKL+11, SW11] introduces π-functions to resolve dependencies between accesses to shared variables in different threads.

Definition 25. *A π-function is introduced for a shared variable x immediately before its use (read access), and has the form $\pi(x_1, \ldots, x_l)$, where each x_i, $1 \leq i \leq l$ is either the last definition (write access) of x in the same thread as the use, or a definition (write access) of x in another thread.*

Converting a trace such as σ into CSSA form denoted by $\mathsf{cssa}(\sigma)$ consists of the following steps:

1. Convert the trace into SSA from (as described in Section 6.1.1).
2. For each use (read access) of a shared variable $x \in \mathbb{G}$, create a unique name such as x' and add the assignment $x' := \pi(x_1, \ldots, x_l)$ before the use of x. Replace x then with the new definition x'.

Thread 1	Thread 2
① $\mathsf{balance}_1 := \mathsf{bal}_1 + 10;$	② $\mathsf{balance}_2 := \mathsf{bal}_1 + 10;$
③ $\mathsf{balance}_3 := \pi(\mathsf{balance}_1, \mathsf{balance}_2);$ ④ $\mathsf{assert}(\mathsf{balance}_3 = \mathsf{bal}_1 + 20);$	

Figure 6.13: Trace with hazard and a π-function

Figure 6.13 shows a trace with two threads with an assignment containing a π-function (arbitrating between the definitions $\mathsf{balance}_1$ and $\mathsf{balance}_2$) inserted before the assertion which accesses balance.

To encode WAR and WAW dependencies, we introduce an irreflexive, transitive, and anti-symmetric relation $\mathsf{hb}(e_i, e_j)$ which indicates that event e_i occurred before event e_j. This happens-before relation enables us to encode the program order and the schedule. In addition, $\mathsf{rd}(\mathsf{x}, e_i)$ and $\mathsf{wr}(\mathsf{x}, e_j)$ indicate that x is read at the occurrence of event e_i and written at the occurrence of e_j. These primitives allow for an explicit encoding of data-dependencies:

$$\begin{array}{rcl} \mathsf{wr}(\mathsf{x}, e_i) \wedge \mathsf{hb}(e_i, e_j) \wedge \mathsf{rd}(\mathsf{x}, e_j) & \Leftrightarrow & \mathsf{raw}_\mathsf{x}(e_i, e_j) \\ \mathsf{rd}(\mathsf{x}, e_i) \wedge \mathsf{hb}(e_i, e_j) \wedge \mathsf{wr}(\mathsf{x}, e_j) & \Leftrightarrow & \mathsf{war}_\mathsf{x}(e_i, e_j) \\ \mathsf{wr}(\mathsf{x}, e_i) \wedge \mathsf{hb}(e_i, e_j) \wedge \mathsf{wr}(\mathsf{x}, e_j) & \Leftrightarrow & \mathsf{waw}_\mathsf{x}(e_i, e_j) \end{array} \tag{6.3}$$

Hazard-sensitive Encoding

The hazard-sensitive encoding presented below incorporates inter-thread data-dependencies into the encoding of a trace. To construct the hazard-sensitive encoding of a trace σ, we convert the trace into CSSA form ($\mathsf{cssa}(\sigma)$) by annotating the events which contain read accesses of shared variables with corresponding π-functions. A π-function for the read access of a shared variable x, takes as parameters the subscripted variables representing the last definition of x in the same thread as the use and all definitions of x in other threads (Definition 25).

Let $\mathsf{cssa}(\sigma)[i]$ contain a read access to a shared variable x and let the corresponding π-function assign subscripted variable x_i: $\mathsf{x}_i = \pi(\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_l)$. If $\mathsf{cssa}(\sigma)[j]$ is an assignment to x_j and the last event before $\mathsf{cssa}(\sigma)[i]$ which updates the shared variable x, then the π-function assigns x_j to x_i. In order to reflect inter-thread data-dependencies between the events $\sigma[i]$ and $\sigma[j]$ and other definitions of x in σ which occur before or after these two events, $\Phi_{\mathsf{hs}}(\sigma[i])$ adds the following formula:

$$\mathrm{PI}(\mathsf{x}, \sigma[i]) \stackrel{\mathrm{def}}{=} \mathsf{rd}(\mathsf{x}, \sigma[i]) \wedge (\mathrm{DEP}(\sigma[i], \sigma[j]) \Rightarrow (\mathsf{x}_i = \mathsf{x}_j)) \tag{6.4}$$

where DEP$(\sigma[i], \sigma[j])$ is the following condition:

$$\mathsf{raw}_{\mathsf{x}}(\sigma[j], \sigma[i]) \wedge \bigwedge_{m \in \{e \in \sigma \,|\, \mathsf{wr}(\mathsf{x}_k, e), 1 \leq k \leq l \wedge k \neq j\}} (\mathsf{waw}_{\mathsf{x}}(m, \sigma[j]) \vee \mathsf{war}_{\mathsf{x}}(\sigma[i], m)) \quad (6.5)$$

where x_k is a parameter of π-function in $\mathsf{x}_i = \pi(\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_l)$, and $\mathsf{x}_k \neq \mathsf{x}_j$.

Intuitively, DEP$(\sigma[i], \sigma[j])$ states that x_j is written before x_i is read, and no other definition of x interferes.

Using Formula (6.4), we define the hazard-sensitive encoding of elements of $\mathsf{cssa}(\sigma)$ as follows:

$$\Phi_{\mathsf{hs}}(\sigma[i]) \stackrel{\text{def}}{=} \Phi(\sigma[i]) \wedge \begin{cases} \mathsf{true} & \text{if } \sigma[i] \text{ accesses no shared var.} \\ \bigwedge_{1 \leq k \leq n} (\mathrm{PI}(\mathsf{x}_k, \sigma[i])) & \text{if } \sigma[i] \text{ reads shared var. } \mathsf{x}_k \\ \bigwedge_{1 \leq k \leq n} (\mathrm{PI}(\mathsf{x}_k, \sigma[i])) \wedge \mathsf{wr}(\mathsf{y}, \sigma[i]) & \text{if } \sigma[i] \text{ reads shared var. } \mathsf{x}_k \text{ and writes shared var. } \mathsf{y} \end{cases} \quad (6.6)$$

where $1 \leq i \leq |\sigma|$ and $\Phi(\sigma[i])$ is defined similar to (6.1) except that here we use $\mathsf{cssa}(\sigma)$ instead of $\mathsf{ssa}(\sigma)$.

Finally, $\Phi_{\mathsf{hs}}(\sigma)$ is a sequence of conjuncts each of which corresponds to encoding of individual elements in $\mathsf{cssa}(\sigma)$. Since the encoding of inter-thread data-dependencies require the hb relation between the events, we insert hb predicates encoding the happens-before relation between every two consecutive events of σ in $\Phi_{\mathsf{hs}}(\sigma)$. The following sequence of conjuncts is then used for computing interpolant sequence:

$$\Phi_{\mathsf{hs}}(\sigma[1]) \wedge \mathsf{hb}(\sigma[1], \sigma[2]) \wedge \Phi_{\mathsf{hs}}(\sigma[2]) \wedge \ldots \wedge \Phi_{\mathsf{hs}}(\sigma[n-1]) \wedge \mathsf{hb}(\sigma[n-1], \sigma[n]) \wedge \Phi_{\mathsf{hs}}(\sigma[n]), \quad (6.7)$$

where $n = |\sigma|$.

Applying inductive sequence interpolation to Formula (6.7) yields a sequence in_1, $\mathsf{out}_1, \ldots, \mathsf{in}_n, \mathsf{out}_n$ of formulas such that

$$\mathsf{in}_i \wedge \Phi_{\mathsf{hs}}(\sigma[i]) \Rightarrow \mathsf{out}_i \text{ and } \mathsf{out}_i \wedge \mathsf{hb}(\sigma[i], \sigma[i+1]) \Rightarrow \mathsf{in}_{i+1}\,.$$

Unlike before, in_i and out_i propagate facts about states as well as execution order. We can slice subtrace $\sigma[i, j]$ if $\mathsf{in}_i \Rightarrow \mathsf{out}_j$, subtrace $\sigma(i, j)$ if $\mathsf{out}_i \Rightarrow \mathsf{in}_j$, subtrace $\sigma[i, j)$ if $\mathsf{in}_i \Rightarrow \mathsf{in}_j$, and subtrace $\sigma(i, j]$ if $\mathsf{in}_i \Rightarrow \mathsf{out}_j$. The resulting sliced trace η corresponds to a sequence of events as well as a set of hb predicates representing context switches and program order constraints relevant to the failure.

Definition 26 (Hazard-sensitive slice)**.** *Given a failing trace σ, a (sound) slice η is* hazard-sensitive *if for every event $\eta[k] = \sigma[i]$ and event $\sigma[j]$ such that there is an inter-thread data-dependency between $\sigma[j]$ and $\sigma[i]$, there exists an event $\eta[h]$ such that $\eta[h] = \sigma[j]$.*

Theorem 6.2.1. *Let σ be a concurrent failing trace and let η be the slice obtained from σ as explained above. Then η is a sound hazard-sensitive slice of σ.*

Proof (sketch). Since the sequence in_1, $\mathsf{out}_1, \ldots, \mathsf{in}_n, \mathsf{out}_n$ used for computing η is an inductive interpolant sequence, according to an argument similar to [MSTC14, Theorem 1], it can be shown that η is sound. It remains to show that η is hazard-sensitive. Assume that $\eta[k] = \sigma[i]$ and there is an inter-thread data-dependency between $\sigma[j]$ and $\sigma[i]$. Let e_i denote $\sigma[i]$ and let e_j denote $\sigma[j]$.

- Assume that e_i is a read access to x, i.e., $\mathsf{rd}(\mathsf{x}, e_i)$. RAW dependencies are readily handled by the SSA encoding. The remaining WAR dependencies are encoded in the π-function introduced for the read access in e_i (which assigns the variable x_i used in e_i).

  Formula (6.5) requires that every event $m \in \sigma$ that writes to x is either visited before the most recent write access to x or after the read access e_i. Assume that $\mathsf{hb}(m, e_j)$ in σ. Then $\mathsf{war}_\mathsf{x}(e_i, m)$ evaluates to false, and the interpolant in_i must imply $\mathsf{wr}(\mathsf{x}, m)$, since otherwise $\mathsf{waw}_\mathsf{x}(m, e_j)$ in the premise (6.5) of Formula (6.4) cannot be discharged. The predicate $\mathsf{wr}(\mathsf{x}, m)$ can only be introduced into the interpolation sequence through $\Phi_\mathsf{hs}(m)$, and therefore event m cannot be sliced away. If $\mathsf{hb}(e_i, m)$ in σ, then the premise of out_i can only be discharged by $\mathsf{wr}(\mathsf{x}, m)$ contributed by $\mathsf{hb}(e_i, m)$. Consequently, if event m is not included, the final interpolant cannot be false.

- Assume that e_i is a write access to x. Then there must also be a relevant read access to x in η. The encoding of the corresponding π-function will enforce that all write accesses conflicting with e_i are included in the slice. This is done via the encoded WAW dependencies in Formula (6.5).

□

Consider the trace in Figure 6.13. A hazard-insensitive slice would contain either the event at ① or the event at ②, but not both. Encoding (6.4) and (6.5) of the events with π-function require the interpolant before ③ to imply $\mathsf{waw}_\mathsf{balance}(①, ②)$, and consequently $\mathsf{wr}(\mathsf{balance}, ①)$ and $\mathsf{wr}(\mathsf{balance}, ②)$. Therefore, events at ① and ② as well as the hb relation $\langle ①, ② \rangle$ are included in the resulting slice.

Case Study: Hazard-Sensitive Slicing Using our new hazard-sensitive encoding, we generated a hazard-sensitive but control-flow-insensitive slice for the case study introduced in Section 6.2.1. Figure 6.14 shows the *full* hazard-sensitive but control-flow-insensitive slice for an initial faulty trace of this case study consisting of 244 instructions. The actual slice consists of 25 instructions that correspond to the 13 program statements shown in Figure 6.14. Notably, the slice now includes the relevant assignments performed by Thread T_1. Statements related to acquiring and releasing locks have been sliced away, since the values of locks are not related to the assertion violation and the slice is control-flow-insensitive. The consecutive updates of balance in T_1 and T_2 included in the slice illustrates the atomicity violation and explains the problem. These updates are now included in the slice due to the encoding of the output dependency (waw) that exists between them.

Note that the slice contains annotations (the interpolants generated by the constraint solver) which aid the understanding. In the hazard-sensitive encoding, Formula (6.6), we use hb, wr, and rd predicates, therefore, they also appear in the computed interpolants. In Figure 6.14, they are not shown since wr, and rd predicates do not carry useful information for understanding the bug, and the ordering information represented by hb predicates can be inferred from the order in which the events of the slice are shown. In the given example, the interpolants reflect the deviation of balance and bal from the expected value. The interpolant before the assertion ($\mathsf{balance} = 60$) is $\mathsf{balance} \leq 10$. The interpolant before and after the interference of thread T_1 in Figure 6.14 indicates that the local variable bal in thread T_2 only accounts for the two withdrawals performed by T_2. Thus, bal holds a stale value. The constraint is unaffected by the assignment to the shared variable balance in thread T_1. This helps the programmer recognize that the deposits performed by T_1 are not reflected in bal, even though all other statements of T_1 are eliminated.

Because of the simple structure of the original traces, the interpolants in our examples are typically conjunctions of inequalities over variables, which are easy to interpret. Interpolants represent *sets* of erroneous states and are therefore generalizations of the single states observed during the execution of the original trace, just as slices are generalizations of the original traces. Note that the slice in combination with the annotations in Figure 6.14 represents a sequence of Hoare triples formally proving that the slice violates the assertion and is therefore sound (cf. Definition 21).

6.2.4 Fine-Tuning Explanations

The encodings presented in Sections 6.2.2 and 6.2.3 can be combined in a straight forward manner. Control-flow or hazard-sensitivity can be added (or removed) by (dis-)regarding scopes and π-functions in the encoding of a failing trace σ. Control-flow dependency can be incorporated into π-nodes in Equation (6.4) by prefixing the assignment $\mathsf{x}_i = \mathsf{x}_j$ with the guard of the definition of x_j at event e_j:

$$\mathsf{guard}(e_j) \Rightarrow (\mathrm{DEP}(e_i, e_j) \Rightarrow (\mathsf{x}_i = \mathsf{x}_j)) \ , \tag{6.8}$$

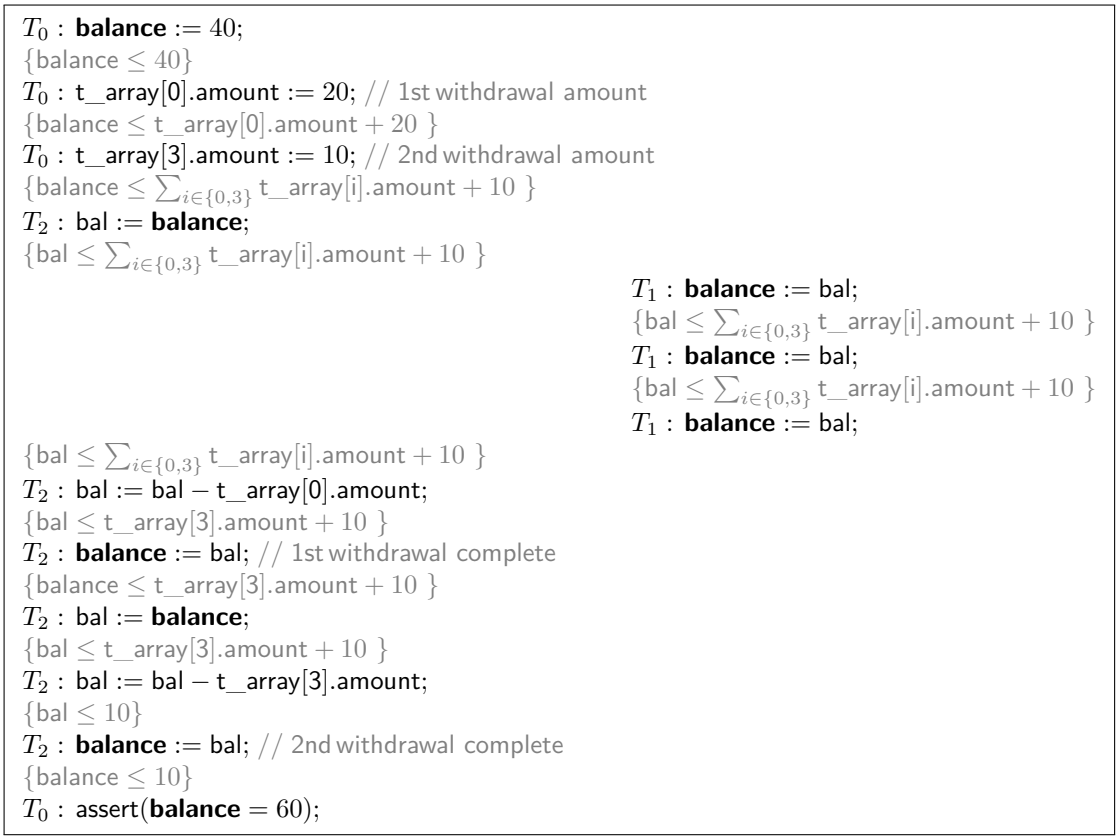


Figure 6.14: Fully annotated hazard-sensitive slice of the trace in Figure 6.6

similar to the guard in the control-flow sensitive encoding of σ as given in Algorithm 6.1. Moreover, Encoding (6.4) can be made insensitive to WAR dependencies by restricting m to predecessors of e_i and by dropping the disjunct $\mathsf{war}_x(e_i, m)$ from (6.5) (and similarly for WAW dependencies). Consequently, our method provides us with a choice of control-, WAR-, and WAW-dependencies reflected by the resulting slices.

Note that flow-dependency has a special role, since use-definition chains are explicit in the SSA representation. The partial order given by the subset relation $\subseteq$ over the power-set of the remaining dependencies $\{\mathsf{cs}, \mathsf{war}, \mathsf{waw}\}$ reflects possible levels of detail in slices, as illustrated by the Hasse diagram in Figure 6.15. As indicated in the diagram, the configuration $\emptyset$ corresponds to the basic approach presented in [ESW12, MSTC14], whereas $\{\mathsf{cs}\}$ represents the control-flow sensitive approach which incorporates both intra- and inter-thread control dependencies.

While we see interpolants as an inherent part of the explanation, the level of detail provided by these annotations cannot be related or formalized as easily as it is the case for dependencies: changing the underlying encoding typically has an unpredictable effect on the structure and strength of interpolants [DPWK10] (since interpolation is not monotone [McM06]).

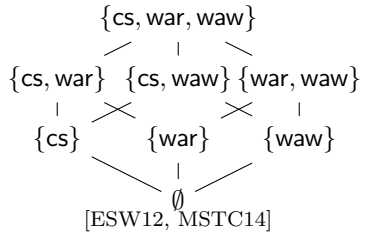


Figure 6.15: Explanations: level of detail

6.3 Experiments

We have implemented our approach as an extension of the directed testing tool ConCrest [FHRV13] and the debugging tool Vermeer [SNOSW15]. We use ConCrest for generating error traces of concurrent programs and then produce slices for bug explanation according to the algorithm presented in Section 6.2.

While all slices provided by our tool are sound in the sense of Definition 21, the results from Section 6.2.4 enable the user to vary the level of detail included in the explanations. However, it is the type and nature of concurrency bugs that determine the level of detail required in order to have the problematic interferences between threads be reflected in the explanation. This section provides an empirical evaluation on the *effectiveness* of slices produced by our method with varying levels of detail in understanding concurrency bugs. Specifically, we discuss in depth how the slice provided by our method reflect a concurrency bug in the implementation of a lock-free concurrent data structure. Finally, we provide an evaluation of how *efficiently* our method can reduce the size of the original traces (in terms of statements, and variables that are included in the explanations).

6.3.1 Effectiveness of the Method

To evaluate the effectiveness of our method, we applied it on a collection of faulty C programs. Table 6.1 summarizes our empirical results. The benchmarks in this table are classified into two groups. The first group consists of 33 multithreaded C programs taken from [KKW15], which combined benchmarks from several sources in the literature. Although these programs are small in terms of the lines of code, they capture the essence of concurrency bugs reported in various versions of open source applications such as Mozilla, Apache, GCC, etc. The apache2 and bluetooth benchmarks in the second group, which are taken from [FHRV13], are also simplified versions of real-world applications with concurrency bugs. The pool-simple-2 is a lock-free concurrent data structure with a linearizability bug. We discuss this benchmark in depth later. The remaining two benchmarks in the second group are two variants of the program discussed in Section 6.2.1. For each benchmark program, the name, the number of lines of code (LOC), the number of threads, and the type of bug are listed in Table 6.1. The table also shows the number of failing traces per benchmark. These numbers vary due to the assertions in the benchmarks and ConCrest's ability to produce failing traces. This variation does not reflect any

preselection of traces. In total, ConCrest generated 90 failing traces from the 38 programs all of which we considered in our evaluation.

The remaining columns in Table 6.1 show how effective the different dependency encodings are at revealing different types of concurrency bugs. We used four different encodings to track data and control dependencies. In the table, **hs** refers to hazard-sensitive encoding for tracking inter-thread data dependencies ($\Phi_{\mathsf{hs}}(\sigma)$, see Section 6.2.3), **cs** refers to control-flow sensitive encoding for tracking control dependencies ($\Phi_{\mathsf{fs}}(\sigma)$ see Sections 6.2.2 and 6.1.2), and **ds** refers to the vanilla SSA encoding ($\Phi(\sigma)$ see Sections 6.1.1) for tracking only flow-dependence between global and local variables, which is implicit in all other encodings. The symbol "+" refers to a combination of different encodings.

Our definition of whether the bug was captured depends on the type of bug. For data race bugs, we required that the slice reflecting the bug contains both conflicting accesses. For atomicity violations, a slice reflecting the bug contains conflicting statements from another thread interrupting the desired atomic region. For order violations, a slice reflecting the bug contains conflicting statements in the problematic order. We use ✓ if the explanations obtained using the corresponding encoding capture the bug, and – to indicate that the bug was not captured.

By manually inspecting the explanations we found that for all but two benchmarks, tracking all dependencies **ds+cs+hs** yields slices that capture the corresponding concurrency bug. However, there exists at least one additional encoding for most benchmarks which results in smaller slices that still reveal the bug. This encoding is usually **hs** (68%) or **cs** (50%) depending on the nature of the bug and the defined assertions. In a few benchmarks such as fibbench, even the vanilla SSA encoding **ds** produces slices which reflect the concurrency bug. The benchmarks hash_table, ms_queue02, and list_seq are the only programs that require the full **ds+cs+hs** encoding. These programs capture bugs in intricate concurrent data structure implementations.

Since tracking more dependencies results in longer slices, we can stop increasing the level of detail in a slice by including more dependencies as soon as the bug is reflected in that explanation. For example, the explanation computed by the encoding **ds+hs** for the account benchmark reveals the atomicity violation bug in this benchmark. Therefore, it is not necessary to compute a longer slice using the **ds+cs+hs** encoding.

We realized that the benchmarks boop, freebsd_auditarg and gcc-java-25530 from [KKW15] contain sequential bugs rather than concurrency bugs (although in [KKW15] they are classified as concurrency bugs). As can be seen in Table 6.1, using the vanilla SSA encoding **ds** is sufficient for understanding the sequential bugs in these benchmarks. For the benchmark freebsd_auditarg, in two failing traces the bug is triggered by sequential executions of the three threads. For these two traces, using any of the four encodings results in a slice which reveals the flow of data leading to the assertion violation. However, in one failing trace the bug is triggered due to an interference between two threads. For this trace, only encodings **ds+hs** and **ds+cs+hs** reflect the cause of the assertion violation (numbers (2/3) in columns **ds** and **ds+cs** for this benchmark indicate that

```
treiber_stack_t* ts[2];
void pool_ins(int v) {
  // we assume that v != EMPTY
  int idx = random()%2;
  ts_push(&ts[idx], v);
}


int pool_rem() {
  int idx = random()%2;
  for (int i = 0; i < 2; i++) {
    int v = ts_pop(&ts[(idx+i)%2]);
    if (v != EMPTY) return v;
  }
  return EMPTY;
}
```

```
void thread1() {
  pool_ins(1);
  pool_rem();
}

void thread2() {
  pool_ins(2);
  int v = pool_rem();
  assert(v != EMPTY);
}
```

Figure 6.16: Faulty pool based on Treiber stacks

these encodings only capture the bug in two out of the three traces).

Only for the two benchmarks apache-25520 and cherokee_01 the slices produced by our method failed to reveal the bugs. The problem is that in these benchmark programs the root cause of the assertion violation is that a specific branch of a conditional statement is not taken during the execution. Our slices currently cannot reveal the non-occurrence of an event as the cause for failure because our analysis focuses on single error traces. In future work, we plan to consider and merge several traces in order to be able to track the effect of alternative branches on the assertion violation. Though, with a simple modification of the assertions that are checked in these two benchmarks, our technique is able to capture the atomicity violations.

We were able to apply our method on 33 out of all 34 benchmark programs used in [KKW15]. The only missing benchmark is fibbench_longer which is a variant of fibbench with larger parameters. For this benchmark, the concurrency testing tool ConCrest [FHRV13] we used failed to generate a failing trace. This specific benchmark exhibits the worst-case behavior for the search heuristic implemented in ConCrest. Though, we would like to point out that our slicing technique is not tight to any specific testing tool and we could deploy other means for finding failing traces.

Running times We found that the generation of the slices is in general very fast with an average of 2.43s ($\sigma = 11.02s$) across all encodings and a maximum of 168.8s. As expected, the running times increase with the amount of detail captured by the encoding. Generating a **ds** slice takes 0.43s on average ($\sigma = 0.18s$) whereas a **ds+cs+hs** slice takes 7.3s ($\sigma = 21.25s$).

Case Study: Lock-free Concurrent Data Structure Benchmark bankaccount_lock-_for_loop in Table 6.2 corresponds to the example discussed in Section 6.2.1. The slice presented there has been generated by our tool. In the following, we further discuss

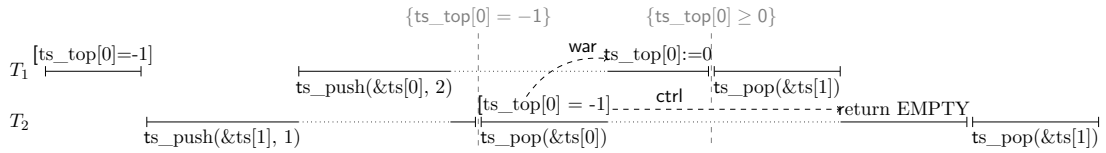


Figure 6.17: Faulty execution of program in Listing 6.16 with dependencies ([] denote conditions)

benchmark pool_simple_2 in more depth as it demonstrates that, in general, both control and hazard-sensitive information is needed to obtain useful bug explanations.

Benchmark pool_simple_2 was provided by Andreas Haas at University of Salzburg, as a real-world example of a linearizability bug in concurrent data structures. It comprises a faulty implementation of a concurrent data structure that stores objects in a pool. Listing 6.16 shows a simplified version of the actual source code that we analyzed. In order to reduce contention, objects that are inserted into this pool are stored in two different stacks $ts[0]$ and $ts[1]$. Each time pool_ins is called, a stack will be picked randomly and the passed value will be stored in the selected stack. Thereby, the amount of conflicting operations from different threads at each concurrent data structure is reduced. In order to further reduce contention, one can add more stacks.

The pool_rem operation of the pool may incorrectly return the designated value EMPTY although the pool is not empty (checked via the assertion in thread2). The problem can occur when pool_rem is called and, for example, stack $ts[1]$ is empty but $ts[0]$ is not. Figure 6.17 shows a corresponding faulty program execution. We describe the explanation our tool provides for one of the faulty traces generated for the pool example. To highlight the problematic dependencies in the execution, we need to inspect the trace at instruction level, as the interferences are not reflected at the level of the overlapping procedure calls. The implementation of the treiber_stack data-structure uses the entry ts_top[i] to store the index of the top element of the i^{th} stack. The value of ts_top[i] is -1 if the corresponding stack is empty. The write access to the actual stack is implemented using an atomic *compare-and-swap* operation (guaranteeing exclusive access to the top of the stack), which only succeeds if no other thread interferes with the write operation. As shown in Listing 6.16, pool_rem iterates over all stacks to check whether one of them contains an element that can be removed.

In the generated trace, the assertion that ts_top[i] must be -1 for all stacks if the pool is reported to be empty fails. The statements in Figure 6.17 are part of the slice reported by our tool and highlight the underlying problem: thread T_1 pushes an element onto stack 0 (ts_top[0]:=0) *after* thread T_2 has determined that the stack is empty. This is captured by the anti-dependency between the statements [ts_top[0]=-1] and ts_top[0]:=0 (denoted by the war edge). Thread T_1 then proceeds to remove the element previously pushed by T_2 onto stack 1. Consequently, thread T_2 finds stack 1 empty and reports that the pool is empty (based on a stale value of ts_top[0]), even though stack 0 still contains one element. This is captured by the control-dependency between [ts_top[0]=-1] and return EMPTY (denoted by the ctrl edge). Thus, even though the assignment ts_top[0]:=0 is

implemented as an atomic compare-and-swap operation in the actual code, this does not guarantee correctness of the lock-free implementation: the operation pool_rem is not *linearizable*, since its effect is not instantaneous.

The core of the problem is accurately reflected by the control-sensitive slice generated by our tool: return EMPTY is necessary to satisfy the premise of the assertion, and ts_top[0]:=0 must be included to contradict the conclusion. The return statement is control-dependent on [ts_top[0] = -1], and the explanation therefore includes the initialization of ts_top[0].

While the control-sensitive slice that our tool computes does *not* explicitly include the assumption [ts_top[0] = -1], the condition is reflected by the error invariant $\mathsf{ts\_top}[0] = -1$. This information is explicit in the hazard-sensitive slice generated by our tool, which includes the anti-dependent statements [ts_top[0] = -1] in thread T_2 and ts_top[0]:=0 in thread T_1. Notably, the control and hazard-sensitive slice is only marginally longer than the control-sensitive slice: the former contains 264 instructions, whereas the latter contains 255 instructions, or 28% of the 924 instructions of the original trace. In addition, our tool drops roughly 44% of the variables of the original trace.

6.3.2 Quantitative Evaluation

Table 6.2 shows the effect of tracking different dependencies on the size of the slices. μ refers to average percentage reduction as the quotient of the number of remaining and original instructions, so smaller numbers mean smaller slices. As expected, increasing the sensitivity of the algorithm by tracking more dependencies tends to lead to more detailed slices, and hence smaller reductions. However, as we have seen previously, the hazard-sensitive slices (**ds+hs**), which capture the concurrency bugs in 68% of the benchmarks, on average contain 35% of the original instructions and 54% of the original variables. We gained the maximum reduction with the vanilla data-sensitive encoding (**ds**), however the resulting explanations reflected the concurrency bugs in only 23% of the benchmarks. The amount of reduction is different across benchmarks with a maximum of 93% for the apache2 benchmark program. As we saw in the bank account example, slices which are hazard sensitive but not control-flow sensitive tend to be much smaller than slices which are control-flow sensitive, but not data-hazard sensitive. In general, we expect that control-flow sensitive slices tend to include more of the local variables of the relevant threads, whereas the hazard sensitive slices focus on shared variables that have inter-thread hazards.

6.4 Related Work

The use of interpolation and error invariants for the purpose of error explanation [ESW12, CESW13, MSTC14] is discussed in some detail in Section 6.1.2. Murali et al. [MSTC14] relates interpolation-based localization to a consistency-based analysis using unsatisfiable cores.

The approach most closely related to ours is implemented in ConcBugAssist [KKW15]. Similar to BugAssist [JM11], which only targets sequential programs, this tool computes a bounded unwinding of the program (generated with CBMC [CKL04]) in an SSA formula and then uses a MAX-SAT solver to localize the faults in the encoded traces. Thus, both BugAssist and ConcBugAssist take into account multiple traces simultaneously which can yield better accuracy in certain cases (e.g., benchmarks apache-25520 and cherokee_01 in our evaluation). Also, ConcBugAssist provides a mechanism for repairing the localized faults. On the other hand, slices reported by core-guided localization techniques such as [JM11, KKW15] (and [MSTC14]), do not contain branch conditions (or statements explaining why they hold). Moreover, we believe that the additional information provided by our approach in the form of error invariants can further aid the understanding of the produced slices.

On the benchmarks from [KKW15], we have found that ConcBugAssist yields similar reduction ratios as our tool using the **ds+hs** encoding. However, the dependency of ConcBugAssist on a bounded model checker for the constraint generation entails scalability issues. E.g., we ran ConcBugAssist on pool_simple_2. Even after simplifying the test harness and manually determining the minimal unwinding depth to detect the bug, the tool timed out after 45 minutes–still in the constraint generation phase. On the other hand, our directed testing tool was able to generate a failing trace with the original test harness in less than 2 minutes and the subsequent slice generation took at most 34s. We observed the same issues with ConcBugAssist on the other additional benchmarks that we considered. In general, the bounded model checker used by ConcBugAssist appears to struggle with benchmarks that involve arrays and non-trivial loops.

Other static approaches for simplifying and summarizing concurrent error traces include [GHR+15], [HZ11], [JS10], and [KG08]. In [GHR+15], an SMT solver is used to derive a symbolic representation of the happens-before relation of all reorderings of a given trace that violate a safety property. The authors then show how to infer bug summarizations from the calculated happens before formula. Our approach differs in the way SMT solvers are deployed for bug explanation. We use interpolation to extract explanations from a proof that a bad trace violates the property. Our encoding of the trace ensures that the proof explicitly captures which happens before relations are relevant for the faulty behavior. On the other hand, [GHR+15] uses model enumeration to calculate all bad reorderings of a given trace and then computes bug summaries with inference rules that capture specific types of concurrency bugs.

SimTrace [HZ11] and Tinertia [JS10] attempt to minimize the number of context switches in a trace by reordering independent statements of concurrent threads. This technique is orthogonal to the approach presented in this paper.

A large number of techniques are proposed for the exposure and detection of concurrency bugs such as race conditions or atomicity/serializability violations. These techniques have in common that they are geared towards specific bug characteristics [FQ03, XBH05, LPSZ08]. Our formal approach does not rely on specific bug characteristics, hence providing a general framework for concurrency bug explanation.

Dynamic techniques such as Falcon [PVH10] and Unicorn [PVH12] detect single- and multivariable atomicity violations as well as order violations by using the given bug pattern templates. Since in our approach we encode data-dependencies, we do not use any given bug pattern templates.

The method of Chapter 5 provides a general framework for explaining concurrency bugs that does not rely on templates. However, the mining method of Chapter 5 for isolating the relevant parts of traces to the error requires failing as well as passing traces. The interpolation-based technique of this chapter only considers failing traces.

Other tools for automatically fixing concurrency-related errors include Afix [JSZ+11] and ConcurrencySwapper [CHR+13]. The latter uses error invariants to generalize a linear error trace to a partially ordered trace. The resulting trace is then used to synthesize a fix. This approach may potentially benefit from our more fine-tuned trace encoding that enables error invariants to capture concurrent data dependencies.

6.5 Summary

In this chapter, we presented a formal framework to generate concise explanations of concurrency bugs, enabling programmers to quickly isolate and focus on the relevant aspects of error traces. By generalizing existing interpolation-based techniques to include data-dependency and hazards, the explanations generated by our approach can capture a broad range of concurrency bugs. We proved that the reported slices are sound and sufficient to trigger the failure. The experimental evaluation of our prototype implementation showed that the approach is effective and significantly reduces the amount of code that needs to be inspected.

Benchmark	**#T**	LOC		AIT	Threads	Bug	**ds+cs+hs**	**ds+hs**	**ds+cs**	**ds**
account	3	43	(58)	51.7	4	AV	✓	✓	–	–
apache-21287	2	30	(79)	43	3	AV	✓	–	✓	–
apache-25520	1	88	(192)	34	3	AV	–	–	–	–
barrier_vf_false	12	57	(85)	27	4	AV	✓	–	✓	–
boop	1	58	(98)	40	3	SB	✓	✓	✓	✓
cherokee_01	1	88	(188)	28	3	AV	–	–	–	–
counter_seq	1	28	(41)	29	3	DR	✓	✓	–	–
fibbench	2	34	(47)	34	3	AV	✓	✓	✓	✓
freebsd_auditarg	3	52	(104)	37	4	SB	✓	✓	✓ (2/3)	✓ (2/3)
gcc-java-25530	2	36	(86)	17	3	SB	✓	✓	✓	✓
gcc-libstdc++-3584	1	40	(104)	37	3	AV	✓	✓	–	–
gcc-libstdc++-21334	1	36	(86)	27	3	OV	✓	✓	–	–
gcc-libstdc++-40518	2	40	(104)	23	3	AV	✓	–	✓	–
glib-512624_02	2	50	(94)	27.5	3	AV	✓	✓	–	–
hash_table	1	51	(114)	69	3	AV	✓	–	–	–
jetty-1187	1	24	(98)	26	3	AV	✓	✓	–	–
lazy01_false	2	39	(55)	23	4	OV	✓	✓	✓	✓
lineEq_2t_01	1	35	(58)	52	3	AV	✓	✓	✓	✓
linux-iio	1	54	(87)	55	3	DR	✓	✓	–	–
linux-tg3	1	93	(115)	167	3	DR	✓	✓	✓	–
list_seq	1	59	(122)	53	3	AV	✓	–	–	–
llvm-8441	2	149	(244)	32.5	3	AV	✓	✓	–	–
mozilla-61369	1	19	(68)	6	1	OV	✓	✓	✓	✓
ms_queue02	1	67	(97)	66	3	AV	✓	–	–	–
mysql5	1	21	(27)	28	3	AV	✓	✓	–	–
mysql-644	1	68	(165)	16	3	AV	✓	✓	–	–
mysql-3596	1	30	(83)	6	3	DR	✓	✓	✓	✓
mysql-12848	1	51	(142)	14	2	AV	✓	–	✓	–
read_write_false	1	78	(140)	58	5	AV	✓	✓	✓	✓
reorder2_false	8	50	(105)	10.5	5	AV	✓	✓	✓	✓
testconc2	1	15	(19)	9	2	AV	✓	✓	–	–
transmission-1.42	1	25	(78)	5	3	DR	✓	✓	✓	✓
VectPrime02	1	97	(183)	115	3	AV	✓	✓	–	–
apache2	8	719	(–)	235.5	3	AV	✓	–	✓	–
bankaccount-lock-for-loop	5	103	(–)	247	3	AV	✓	✓	–	–
bankaccount-simple-lock	2	50	(–)	45	3	AV	✓	✓	–	–
bluetooth	5	87	(–)	35.8	3	AV	✓	–	✓	–
pool-simple-2	8	298	(–)	885.5	3	LV	✓	–	✓	–
Total	90						88	47	61	22

ds: Basic Encoding
cs: Control-Sensitive Encoding
hs: Hazard-Sensitive Encoding
#T: No. of Traces in Benchmark
LOC: Lines of Code[a]
AV: Atomicity Violation
SB: Sequential Bug
DR: Data Race
OV: Order Violation
LV: Linearizability Violation
AIT: Average No. of Instructions in a Trace

Table 6.1: Encodings which result in slices reflecting concurrency bugs.

[a]The lines of code reported are counted with the tool cloc (`https://github.com/AlDanial/cloc`, v1.66) excluding comments and blank lines. The lines of code presented in brackets are as stated in [KKW15].

Benchmark	**#T**	**ds+cs+hs** S[%]		V[%]		**ds+hs** S[%]		V[%]		**ds+cs** S[%]		V[%]		**ds** S[%]		V[%]	
		μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
account	3	**62**	**12**	**77**	**6**	**42**	**10**	**68**	**6**	43	8	68	6	29	5	59	5
apache-21287	2	**72**	**0**	**87**	**0**	28	0	53	0	**51**	**0**	**87**	**0**	9	0	40	0
apache-25520	1	38	–	50	–	9	–	33	–	26	–	50	–	9	–	33	–
barrier_vf_false	12	**70**	**0**	**80**	**0**	19	0	40	0	**67**	**0**	**80**	**0**	15	0	40	0
boop	1	**38**	–	**47**	–	**30**	–	**40**	–	**35**	–	**47**	–	**28**	–	**40**	–
cherokee_01	1	46	–	60	–	11	–	40	–	32	–	60	–	11	–	40	–
counter_seq	1	**72**	–	**90**	–	**38**	–	**70**	–	52	–	80	–	31	–	60	–
fibbench	2	**94**	**3**	**97**	**3**	**94**	**3**	**97**	**3**	**88**	**3**	**97**	**3**	**88**	**3**	**97**	**3**
freebsd_auditarg	3	**67**	**7**	**86**	**0**	**32**	**5**	**64**	**0**	**57**	**10**	**79**	**10**	**30**	**8**	**57**	**10**
gcc-java-25530	2	**35**	**0**	**40**	**0**	**35**	**0**	**40**	**0**	**24**	**0**	**40**	**0**	**24**	**0**	**40**	**0**
gcc-libstdc++-3584	1	**62**	–	**79**	–	**35**	–	**64**	–	46	–	71	–	30	–	57	–
gcc-libstdc++-21334	1	**63**	–	**78**	–	**22**	–	**33**	–	48	–	78	–	15	–	33	–
gcc-libstdc++-40518	2	**43**	**0**	**56**	**0**	30	0	56	0	**39**	**0**	**56**	**0**	22	0	56	0
glib-512624_02	2	**84**	**2**	**100**	**0**	**47**	**3**	**80**	**0**	60	4	85	5	38	5	65	5
hash_table	1	**41**	–	**61**	–	4	–	21	–	29	–	54	–	4	–	21	–
jetty-1187	1	**81**	–	**100**	–	**35**	–	**78**	–	58	–	89	–	27	–	67	–
lazy01_false	2	**91**	**0**	**100**	**0**	**65**	**0**	**100**	**0**	**87**	**0**	**100**	**0**	**61**	**0**	**100**	**0**
lineEq_2t_01	1	**69**	–	**81**	–	**46**	–	**71**	–	**52**	–	**76**	–	**37**	–	**67**	–
linux-iio	1	**40**	–	**59**	–	**20**	–	**50**	–	27	–	41	–	16	–	32	–
linux-tg3	1	**19**	–	**38**	–	**13**	–	**36**	–	**8**	–	**11**	–	2	–	9	–
list_seq	1	**58**	–	**95**	–	6	–	30	–	40	–	75	–	6	–	30	–
llvm-8441	2	**74**	**4**	**92**	**0**	**18**	**0**	**33**	**0**	55	7	83	8	12	0	33	0
mozilla-61369	1	**67**	–	**100**	–	**67**	–	**100**	–	**67**	–	**100**	–	**67**	–	**100**	–
ms_queue02	1	**44**	–	**52**	–	5	–	20	–	35	–	48	–	5	–	20	–
mysql5	1	**82**	–	**89**	–	**46**	–	**67**	–	46	–	89	–	25	–	67	–
mysql-644	1	**38**	–	**33**	–	**38**	–	**33**	–	25	–	33	–	25	–	33	–
mysql-3596	1	**100**	–	**100**	–	**100**	–	**100**	–	**67**	–	**100**	–	**67**	–	**100**	–
mysql-12848	1	**71**	–	**67**	–	43	–	50	–	**50**	–	**67**	–	29	–	50	–
read_write_false	1	**17**	–	**27**	–	**17**	–	**27**	–	**17**	–	**27**	–	**17**	–	**27**	–
reorder2_false	8	**86**	**14**	**100**	**0**	**86**	**14**	**100**	**0**	**62**	**8**	**100**	**0**	**62**	**8**	**100**	**0**
testconc02	1	**89**	–	**100**	–	**89**	–	**100**	–	56	–	100	–	56	–	100	–
transmission-1.42	1	**100**	–	**100**	–	**100**	–	**100**	–	**80**	–	**100**	–	**80**	–	**100**	–
VectPrime02	1	**25**	–	**68**	–	**9**	–	**45**	–	18	–	59	–	7	–	36	–
apache2	8	**8**	**2**	**9**	**2**	1	0	1	0	**7**	**2**	**9**	**2**	1	0	1	0
bankaccount-lock-for-loop	5	**46**	**2**	**44**	**2**	**12**	**1**	**30**	**2**	40	2	42	2	9	1	23	3
bankaccount-simple-lock	2	**71**	**0**	**80**	**0**	**31**	**0**	**60**	**0**	62	0	73	0	24	0	53	0
bluetooth	5	**42**	**0**	**63**	**0**	14	0	31	0	**36**	**0**	**63**	**0**	11	0	31	0
pool-simple-2	8	**30**	**1**	**58**	**2**	0	0	2	0	**29**	**1**	**56**	**2**	0	0	2	0
Total	90	58.8		72		35		54		45		67.7		27		50.5	

#T: No. of Traces in Benchmark
S: Slice Size / Trace Size
μ: Average
ds: Basic Encoding
hs: Hazard-Sensitive Encoding
V: No. of Variables in Slice / No. of Variables in Trace
σ: Standard Deviation
cs: Control-Sensitive Encoding

Table 6.2: Quotient of the number of instructions (variables) in the slice and the original trace

CHAPTER 7

Conclusion and Future Directions

7.1 Summary of the Contributions

This dissertation makes contributions by developing automated approaches to address the problem of explaining concurrency bugs or understanding the cause of failure in concurrent systems. Our work culminated in two different approaches for explaining concurrency bugs both in message passing and shared memory concurrent programs. These two approaches which are anomaly detection and slicing reduce the manual effort involved in debugging concurrent programs. Our techniques based on these two approaches provide a general framework for concurrency bug explanation which do not rely on any characteristics specific to one type of concurrency bug.

To extract anomalies for explaining concurrency bugs, the methods of Chapters 3, 4 and 5 adapt a standard frequent pattern mining algorithm called sequential pattern mining. We refer to the computed anomalies as explanatory sequences in message passing systems (Chapters 3 and 4) and bug explanation patterns in multithreaded shared variable programs (Chapter 5) and argued their effectiveness in understanding the cause of failure in concurrent systems. In Chapter 3, we showed that mining explanatory sequences from lengthy execution traces is intractable due to the combinatorial explosion of the potential candidates. In order to make the problem tractable, we proposed three approximation techniques in Chapters 3–5. The substring mining method of Chapter 3 extracts anomalies containing events which occur consecutively inside counterexamples. Although, anomalies can be computed efficiently using this method, they only reveal fractions of explanatory sequences. The subtrace mining method of Chapter 4 improves our explanation by extracting anomalies which occur as subsequences of the counterexamples. However, anomalies are mined from subtraces which are partitions of original traces. Since subtraces provide local view of the traces, the method of Chapter 4 may fail to explain some bugs. The abstraction method of Chapter 5 addresses the shortcomings of the two previous approximation techniques. It improves scalability by reducing the length and the number

of distinct events in traces using a novel abstraction technique. Moreover, the abstraction method preserves the original patterns because abstract traces preserve the ordering information between events of original traces.

While the mining methods of Chapters 3–5 require logging of a number of failing and passing traces, the slicing method of Chapter 6 analyzes one single failing trace. In Chapter 6, we first showed the inadequacy of dynamic slicing techniques in isolating relevant statements and values to a failure. We then proposed a formal framework based on interpolation for constructing semantics-aware slices. The experimental evaluation showed that the slices computed by our method provide concise explanations for different types of concurrency bugs. Moreover, we achieved a significant reduction in the number of variables and the length of traces using our slicing technique.

7.2 Comparison between the Proposed Approaches

As we have seen, in this dissertation, the two approaches we present for concurrency bug explanation are based on different underlying techniques. The anomaly detection approach is based on mining a set of concrete execution traces while the slicing approach is based on analyzing a single symbolic trace using a proof-based technique. Therefore, we were able to compare the efficiency and effectiveness of these approaches in explaining bugs and to learn their strengths and limitations. Since the slicing technique has been developed for explaining bugs in multithreaded shared variable programs, we compare it with the mining method of Chapter 5 which has the same goal and similar setting.

Although both approaches (methods of Chapter 5 and Chapter 6) are based on dynamic analysis and analyze execution traces, in the mining based method we only log read and write accesses of the shared variables (Section 5.1.2) while in the slicing method we need to log all the atomic instructions (Section 6.1.1). Therefore, the heavier instrumentation of the source code which is required in the slicing technique results in a higher overhead and consequently a higher slowdown at runtime.

In terms of scalability, the mining based technique performed significantly better than than the slicing technique in our experimental evaluation. While our largest benchmark in the slicing method has 298 LOC and traces with less than 1000 events, the mining method could compute effective anomalies for a full application (*bzip2smp*) with 6400 LOC and traces with around 13000 events. Note that the events in the traces of mining method include only read and write accesses of shared variables. Regarding efficiency, we found that both techniques are in general very fast in generating anomalies or slices.

The outputs produced by both techniques need to be inspected by the programmer. However, in our experiments, we found that the manual inspection of anomalies computed by the mining technique can be done easier and faster than the manual inspection of the slices produced by the slicing technique. In all case studies, we were able to understand the cause of failure by inspecting the data-dependencies between at most four macros (Figure 5.5).

A failing trace in the slicing technique is defined as a feasible trace reaching a failing assertion (Section 6.1.1) and the failing assertion appears in the unsatisfiable formula which is used for constructing the slice. Therefore, the computed slice depends on the variables that appear in the assertion. If the variables used in the definition of the assertion are not chosen carefully, it can be the case that the bug is not captured by the slice. However, the anomalies computed by the mining method are independent of the defined assertions and reflect the anomalous behaviors that are frequent in the failing traces.

However, we proved that the slices computed by the slicing technique are sound while the mining method may produce false positives (although we did not get false positives in our experiments). Moreover, the quality of the failing and passing datasets affect the computed anomalies in the mining method. On the other hand, the slicing technique analyzes one single failing trace and always generates a sound slice which is sufficient to trigger the failure. For the mining method we lack a formal reasoning regarding the soundness and completeness.

7.3 Future Directions

The techniques presented in this dissertation can be enhanced in terms of efficiency and effectiveness which we plan to do as future work. Moreover the new challenges that we faced during the development of these techniques can form the basis for the future work. Currently, no classification and clustering are done on the anomalies generated by the mining method. The effectiveness of the bug explanation patterns can be improved by first clustering them according to the likelihood that they indicate the same bug. Second, by classifying them according to the type of the concurrency bug that they reflect. The quality of the computed anomalies can be further improved by considering a similarity measure in generating failing and passing datasets. It is also desirable to study the correlation between the similarity of traces in the failing and passing datasets and the number of the different bugs that are reflected in the final result set. We plan to investigate the possibility of making our mining based method *online* for analyzing the traces as they are being generated in order to reduce the overhead of logging traces for offline processing. Moreover, we believe that our abstraction technique (Chapter 5) has potential application in the other domains where runtime information is being processed and analyzed such as runtime verification.

Since the mining method of Chapter 5 based on abstraction of traces has been formalized in the setting of multithreaded shared variable programs, we plan to adapt this technique for explaining bugs in message passing concurrent systems as well.

To increase the accuracy of our slicing technique for bugs that are caused by the absence of an event, we plan as future work to explore techniques for merging multiple related traces. For example, if a bug occurs because a specific branch is not taken at runtime, our slicing method is not able to capture that (See Section 6.3.1). This is because we only analyze one single trace in which the specific branch that has to be taken in

order to avoid the failure is missing. This problem can be solved by considering traces taking alternative branches in our encoding. Moreover, we will incorporate abstraction techniques for summarizing threads and procedure calls in order to further improve the reduction rates for long traces.

List of Figures

List of Tables

List of Algorithms

Bibliography

[ACN+01] Bowen Alpern, Jong-Deok Choi, Ton Ngo, Manu Sridharan, and John M. Vlissides. A perturbation-free replay platform for cross-optimized multithreaded applications. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01), San Francisco, CA, April 23-27, 2001*, page 23, 2001.

[AHB03] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. In *New Technologies for Information Systems, Proceedings of the 3rd International Workshop on New Developments in Digital Libraries, NDDL 2003, and the 1st International Workshop on Validation and Verification of Software for Enterprise Information Systems, VVEIS 2003, In conjunction with ICEIS 2003, Angers, France, April 2003*, pages 82–93, 2003.

[AHLW95] Hiralal Agrawal, Joseph R. Horgan, Saul London, and W. Eric Wong. Fault localization using execution slices and dataflow tests. In *Sixth International Symposium on Software Reliability Engineering, ISSRE 1995, Toulouse, France, October 24-27, 1995*, pages 143–151, 1995.

[Ali12] Mohammad Amin Alipour. Automated fault localization techniques; a survey. Technical report, School of Electrical Engineering and Computer science, Oregon State University, 2012.

[AS95] R. Agrawal and R. Srikant. Mining sequential patterns. In *11th International Conference on Data Engineering(ICDE'95)*, 1995.

[BAEFU06] Yosi Ben-Asher, Yaniv Eytani, Eitan Farchi, and Shmuel Ur. Producing scheduling that causes concurrent programs to fail. In *Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging*, PADTAD '06, pages 37–40, New York, NY, USA, 2006. ACM.

[BBDC+09] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard Trefler. Explaining counterexamples using causality. In *Proceedings of CAV 2009, LNCS. Springer*, 2009.

[BH03] David Binkley and Mark Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *19th International*

Conference on Software Maintenance (ICSM 2003), The Architecture of Existing Systems, 22-26 September 2003, Amsterdam, The Netherlands, pages 44–53, 2003.

[BJGC13] Tom Britton, Lisa Jeng, and Tomer Katzenellenbogen Graham Carver, Paul Cheak. Reversible debugging software. Technical report, Judge Business School, University of Cambridge, 2013.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, Cambridge, Massachusetts, 2008.

[BKMN10] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 167–178, New York, NY, USA, 2010. ACM.

[BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 211–230, New York, NY, USA, 2002. ACM.

[BNR03] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2003.

[BR02] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL 02: Principles of Programming Languages, ACM*, 2002.

[bzi] `http://bzip2smp.sourceforge.net/`, (bzip2smp 1.0), accessed in September 2015.

[CER] Cert/cc advisories. http://www.cert.org/advisories/.

[CESW13] Jügen Christ, Evren Ermis, Matthias Schaef, and Thomas Wies. Flow-sensitive fault localization. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, 2013.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, volume 1855 of *LNCS*, pages 154–169, 2000.

[CGP99] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, December 1999.

[CHR+13] Pavol Cerný, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. Efficient synthesis for concurrency by semantics-preserving transformations. In *CAV*, volume 8044 of *LNCS*, pages 951–967. Springer, 2013.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In *TACAS*, pages 168–176. Springer, 2004.

[CZ02] Jong-Deok Choi and Andreas Zeller. Isolating failure-inducing thread schedules. *SIGSOFT Softw. Eng. Notes*, 27(4):210–220, July 2002.

[DGR04] Nelly Delgado, Ann Q. Gates, and Steve Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering (TSE)*, 30(12):859–872, 2004.

[dJR10] Marc de Jonge and Theo Ruys. The SpinJa model checker. In *Model Checking Software, LNCS. Springer*, 2010.

[DLZ05] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In *Proceedings of the 19thj European Conference on Object-Oriented Programming*, 2005.

[DP07] Guozhu Dong and Jian Pei. *Sequence Data Mining*. Springer, 2007.

[DPWK10] Vijay D'Silva, Mitra Purandare, Georg Weissenbacher, and Daniel Kroening. Interpolant strength. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 5944 of *LNCS*, pages 129–145. Springer, 2010.

[EA03a] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.

[EA03b] Dawson R. Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–252. ACM, 2003.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[EL02] Thomas Eiter and Thomas Lukasiewicz. Complexity results for structure-based causality. *Artif. Intell.*, 142(1):53–89, November 2002.

[EMBO10] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 151–162. USENIX Association, 2010.

[EQT10] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race-aware Java runtime. *Communications of the ACM*, 53(11):85–92, 2010.

[ESW12] Evren Ermis, Martin Schäf, and Thomas Wies. Error invariants. In *Proceedings of the 18th International Symposium on Formal Methods (FM)*, pages 187–201, 2012.

[FF04] Cormac Flanagan and Stephen N Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 256–267, New York, NY, USA, 2004. ACM.

[FF10] Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. *Communications of the ACM*, 53(11):93–101, 2010.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 110–121, New York, NY, USA, 2005. ACM.

[FHRV13] Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. Con2colic testing. In *Foundations of Software Engineering (FSE)*, pages 37–47. ACM, 2013.

[FKS+08] Scott D. Fleming, Eileen Kraemer, R. E. Kurt Stirewalt, Shaohua Xie, and Laura K. Dillon. A study of student strategies for the corrective maintenance of concurrent software. In *International Conference on Software Engineering (ICSE)*, pages 759–768. ACM, 2008.

[FLS06] G. D. Fatta, S. Leue, and E. Stegantova. Discriminative pattern mining in software fault detection. In *Proceedings of the 3rd international workshop on Software quality assurance*, 2006.

[FQ03] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI*, pages 338–349. ACM, 2003.

[GCKS06] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error explanation with distance metrics. In *International Journal on Software Tools for Technology Transfer (STTT)*, 2006.

[GHR+15] Ashutosh Gupta, Thomas A. Henzinger, Arjun Radhakrishna, Roopsha Samanta, and Thorsten Tarrach. Succinct representation of concurrent trace sets. In *POPL*, pages 433–444. ACM, 2015.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[GN08] Patrice Godefroid and Nachiappan Nagappan. Concurrency at microsoft – an exploratory survey. Technical report, Microsoft Research, 2008.

[God97] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 174–186, New York, NY, USA, 1997. ACM.

[Goe03] B. Goethals. Survey on frequent pattern mining, 2003. manuscript.

[GV03] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In *Model Checking Software, LNCS. Springer*, 2003.

[Hal04] Ned Hall. Two concepts of causation. In John Collins and Ned Hall and Laurie Paul, editor, *Causation and Counterfactuals*, pages 225–276. The MIT Press, 2004.

[Hal15] Joseph Y. Halpern. A modification of the halpern-pearl definition of causality. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 3022–3033, 2015.

[HDVT08] Christian Hammer, Julian Dolby, Mandana Vaziri, and Frank Tip. Dynamic detection of atomic-set-serializability violations. In *International Conference on Software Engineering (ICSE)*, pages 231–240. ACM, 2008.

[HFGO94] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[HL02] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 291–301, New York, NY, USA, 2002. ACM.

[Hol03] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addision-Wesley, 2003.

[HP00] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA pathfinder. *STTT*, 2(4):366–381, 2000.

[HP05] Joseph Y. Halpern and Judea Pearl. Causes and explanations: A structural-model approach. part I: Causes. In *The British Journal for the Philosophy of Science*, pages 843–887, 2005.

[HS08] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.

[HSNTB+16] Andreas Holzer, Daniel Schwartz-Narbonne, Mitra Tabaei-Befrouei, Georg Weissenbacher, and Thomas Wies. Error invariants for concurrent traces. In *Proceedings of the 21th International Symposium on Formal Methods (FM)*, 2016.

[HZ11] Jeff Huang and Charles Zhang. An efficient static trace simplification technique for debugging concurrent programs. In *Static Analysis Symposium (SAS)*, volume 6887 of *LNCS*, pages 163–179. Springer, 2011.

[JHS02] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *ICSE*, pages 467–477. ACM, 2002.

[JM11] Manu Jose and Rupak Majumdar. Cause clue clauses: Error localization using maximum satisfiability. *SIGPLAN Not.*, 46(6):437–446, June 2011.

[JPPS11] Nicholas Jalbert, Cristiano Pereira, Gilles Pokam, and Koushik Sen. Radbench: A concurrency bug benchmark suite. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, 2011.

[JS10] Nicholas Jalbert and Koushik Sen. A trace simplification technique for effective debugging of concurrent programs. In *Foundations of Software Engineering (FSE)*, pages 57–66. ACM, 2010.

[JSZ+11] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *PLDI*, pages 389–400. ACM, 2011.

[JTLL10] Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 241–255, New York, NY, USA, 2010. ACM.

[KCL04] D. Kroening, E. Clarke, and F. Lerd. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2004.

[KG08] Sujatha Kashyap and Vijay K. Garg. Producing short counterexamples using "crucial events". In *CAV*, volume 5123 of *LNCS*, pages 491–503. Springer, 2008.

[KKW15] Sepideh Khoshnood, Markus Kusano, and Chao Wang. Concbugassist: constraint solving for diagnosis and repair of concurrency bugs. In *ISSTA*, pages 165–176. ACM, 2015.

[KL00] M. Kamel and S. Leue. Vip: A visual editor and compiler for v-promela. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS 1785, Springer Verlag*, 2000.

[KLL11] Matthias Kuntz, Florian Leitner-Fischer, and Stefan Leue. From probabilistic counterexamples via causality to fault trees. In *Computer Safety, Reliability, and Security - 30th International Conference, SAFECOMP 2011, Naples, Italy, September 19-22, 2011. Proceedings*, pages 71–84, 2011.

[KOESH07] Hermann Kopetz, Roman Obermaisser, Christian El Salloum, and Bernhard Huber. Automotive software development for a multi-core system-on-a-chip. In *Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems*, SEAS '07, pages 2–, Washington, DC, USA, 2007. IEEE Computer Society.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[LAZJ03] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. *SIGPLAN Not.*, 38(5):141–154, May 2003.

[LC09] B. Lucia and L. Ceze. Finding concurrency bugs with context-aware communication graphs. In *Symposium on Microarchitecture (MICRO)*, pages 553–563. ACM, 2009.

[Lew01] David Lewis. *Counterfactuals*. Wiley-Blackwell, 2001.

[LFL13] Florian Leitner-Fischer and Stefan Leue. Causality checking for complex system models. In *Proceedings of 14th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), LNCS, Springer Verlag*, 2013.

[LJZ07] Shan Lu, Weihang Jiang, and Yuanyuan Zhou. A study of interleaving coverage criteria. In *Foundations of Software Engineering (FSE)*, ESEC-FSE companion, pages 533–536. ACM, 2007.

[LKL07] D. Lo, S.C. Khoo, and C. Liu. Efficient mining of iterative patterns for software specification discovery. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2007.

[LM09] Chu Min Li and Felip Manyà. *MaxSAT, Hard and Soft Constraints*, volume 185, pages 613–631. IOS Press, 2009.

[LMC87] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36(4):471–482, 1987.

[LNZ+05] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. *SIGPLAN Not.*, 40(6):15–26, June 2005.

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ACM Sigplan Notices*, volume 43, pages 329–339. ACM, 2008.

[LS05] Mark H. Liffiton and Karem A. Sakallah. On finding all minimally unsatisfiable subformulas. In *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, pages 173–186, 2005.

[LT93] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, July 1993.

[LTB12] S. Leue and M. Tabaei-Befrouei. Counterexample explanation by anomaly detection. In *Model Checking and Software Verification (SPIN)*, 2012.

[LTB13] S. Leue and M. Tabaei-Befrouei. Mining sequential patterns to explain concurrent counterexamples. In *Model Checking and Software Verification (SPIN)*, 2013.

[LTQZ06] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[LWC11] Brandon Lucia, Benjamin P. Wood, and Luis 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, New York, NY, USA, 2011. ACM.

[LYY+05] C. Liu, X. Yan, H. Yu, J. Han, and P. S. Yu. Mining behavior graphs for backtrace of noncrashing bugs. In *Proceedings of the Fifth SIAM International Conference on Data Mining*, 2005.

[Maz86] Antoni W. Mazurkiewicz. Trace theory. In *Petri Nets: Central Models and Their Properties, Advances in Petri Nets*, volume 255 of *LNCS*, pages 279–324. Springer, 1986.

[McM05] Kenneth L. McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.

[McM06] Kenneth L. McMillan. Lazy abstraction with interpolants. In *CAV*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.

[MCT08] Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 289–300, Washington, DC, USA, 2008. IEEE Computer Society.

[ME10] Nizar R. Mabroukeh and C. I. Ezeife. A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys*, 43(1):3:1–3:41, December 2010.

[MOT10] Abdullah Muzahid, Norimasa Otsuki, and Josep Torrellas. Atomtracker: A comprehensive approach to atomic region inference and violation detection. In *IEEE/ACM International Symposium on Microarchitecture (MICRO*, pages 287–297, 2010.

[MQ06] Madan Musuvathi and Shaz Qadeer. CHESS: systematic stress testing of concurrent software. In *Logic-Based Program Synthesis and Transformation (LOPSTR)*, volume 4407 of *LNCS*, pages 15–16. Springer, 2006.

[MQ07] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455. ACM, 2007.

[MQB07] Madanlal Musuvathi, Shaz Qadeer, and Thomas Ball. CHESS: A systematic testing tool for concurrent software. Technical Report MSR-TR-2007-149, Microsoft Research, November 2007.

[MS00] Evan Marcus and Hal Stern. *Blueprints for High Availability.* JohnWilley and Sons, 2000.

[MSTC14] Vijayaraghavan Murali, Nishant Sinha, Emina Torlak, and Satish Chandra. A hybrid algorithm for error trace explanation. In *VSTTE*, 2014.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design Implementation.* Morgan Kaufmann, 1997.

[Nas] Nasdaq's facebook glitch came from 'race conditions'. http://www.computerworld.com/article/2504676/financial-it/nasdaq-s-facebook-glitch-came-from–race-conditions-.html, accessed in November 2015.

[NAW06] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. ACM.

[NAW+08] Syeda Nessa, Muhammad Abedin, W. Eric Wong, Latifur Khan, and Yu Qi. Software fault localization using n-gram analysis. In *Wireless Algorithms, Systems, and Applications, LNCS. Springer*, 2008.

[NM91a] R.H.B. Netzer and B.P. Miller. Improving the accuracy of data race detection. In *Proceedings of the 3rd ACM Symposium on Principles and Practice of Parallel Programming, ACM Press*, 1991.

[NM91b] Robert H. B. Netzer and Barton P. Miller. Improving the accuracy of data race detection. *SIGPLAN Notices*, 26(7):133–144, April 1991.

[OC03] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. *SIGPLAN Not.*, 38(10):167–178, June 2003.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 177–184, New York, NY, USA, 1984. ACM.

[Pap79] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[Pel06] Radek Pelanek. Benchmarks for explicit model checkers, 2006. http://anna.fi.muni.cz/models.

[PHMA+01] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *17th International Conference on Data Engineering (ICDE'01)*, 2001.

[PLZ09] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 25–36. ACM, 2009.

[PNK11] S. Parsa, S. Arabi Naree, and N. Ebrahimi Koopaei. Software fault localization via mining execution graphs. In *ICCSA*, 2011.

[Pou04] Kevin Poulsen. Tracking the blackout bug, 2004. http://www.securityfocus.com/news/8412.

[PS92] Hsin Pan and Eugene H. Spafford. Heuristics for automatic localization of software faults. Technical report, Purdue University, 1992.

[PS07] Eli Pozniansky and Assaf Schuster. Multirace: Efficient on-the-fly data race detection in multithreaded c++ programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(3):327–340, March 2007.

[PVH10] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. Falcon: fault localization in concurrent programs. In *International Conference on Software Engineering (ICSE)*, pages 245–254. ACM, 2010.

[PVH12] Sangmin Park, Richard Vuduc, and Mary Jean Harrold. A unified approach for localizing non-deadlock concurrency bugs. In *Software Testing, Verification and Validation (ICST)*, pages 51–60. IEEE, 2012.

[Rei87] R Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, April 1987.

[RFZO12] Jeremias Rößler, Gordon Fraser, Andreas Zeller, and Alessandro Orso. Isolating failure causes through test case generation. In *International Symposium on Software Testing and Analysis*, pages 309–319. ACM, 2012.

[RGJ07] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Path-sensitive inference of function precedence protocols. In *Proceedings of the 29th international conference on Software Engineering(ICSE)*, 2007.

[RR03] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 30–39. IEEE Computer Society, 2003.

[SBN+97a] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *Transactions on Computer Systems (TOCS)*, 15(4):391–411, November 1997.

[SBN+97b] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. In *ACM Transactions on Computer Systems (TOCS), vol. 15, no. 4*, 1997.

[Sen08] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 11–21, New York, NY, USA, 2008. ACM.

[SNOSW15] Daniel Schwartz-Narbonne, Chanseok Oh, Martin Schäf, and Thomas Wies. VERMEER: A tool for tracing and explaining faulty C programs. In *International Conference on Software Engineering (ICSE)*, 2015.

[SW11] Nishant Sinha and Chao Wang. On interference abstractions. In *POPL*, pages 423–434. ACM, 2011.

[TBWW14] Mitra Tabaei-Befrouei, Chao Wang, and Georg Weissenbacher. Abstraction and mining of traces to explain concurrency bugs. In *Proceedings of the 14th International Conference on Runtime Verification (RV)*, 2014.

[TBWW16] Mitra Tabaei Befrouei, Chao Wang, and Georg Weissenbacher. Abstraction and mining of traces to explain concurrency bugs. *Formal Methods in Systems Design (FMSD)*, January 2016.

[Tip95] F. Tip. A survey of program slicing techniques. *JOURNAL OF PROGRAMMING LANGUAGES*, 3:121–189, 1995.

[Val79] L.G. Valiant. *The Complexity of Computing the Permanent.* Theoretical Computer Science, 1979.

[VM05] Willem Visser and Peter C. Mehlitz. Model checking programs with java pathfinder. In *Model Checking Software, 12th International SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005, Proceedings*, page 27, 2005.

[WD09] W. Eric Wong and Vidroha Debroy. A survey of software fault localization. Technical Report UTDCS-45-09, Department of Computer Science, The University of Texas at Dallas, November 2009.

[Wei81] Mark Weiser. Program slicing. In *International Conference on Software Engineering (ICSE)*, pages 439–449. IEEE, 1981.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[WH04] J. Wang and J. Han. Bide: Efficient mining of frequent closed sequences. In *ICDE*, 2004.

[WKL+11] Chao Wang, Sudipta Kundu, Rhishikesh Limaye, Malay K. Ganai, and Aarti Gupta. Symbolic predictive analysis for concurrent programs. *Formal Aspects of Computing*, 23(6):781–805, 2011.

[WS06] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. *TSE*, 32(2):93–110, 2006.

[WWY+14] Wenwen Wang, Chenggang Wu, Pen-Chung Yew, Xiang Yuan, Zhenjiang Wang, Jianjun Li, and Xiaobing Feng. Concurrency bug localization using shared memory access pairs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 375–376, New York, NY, USA, 2014. ACM.

[WYIG06] Chao Wang, Zijiang Yang, Franjo Ivancic, and Aarti Gupta. Whodunit? causal analysis for counterexamples. In *Automated Technology for Verification and Analysis, LNCS. Springer*, 2006.

[XBH05] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, pages 1–14. ACM, 2005.

[Yan04] G. Yang. The complexity of mining maximal frequent itemsets and maximal frequent patterns. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2004.

[Yan06] G. Yang. Computational aspects of mining maximal frequent patterns. In *Theoretical Computer Science, Volume 362, Issues 1-3, Pages 63-85*, 2006.

[YCGK07] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Distributed dynamic partial order reduction based verification of threaded software. In *Model Checking and Software Verification (SPIN)*, pages 58–75. LNCS, 2007.

[YHA03] X. Yan, J. Han, and R. Afshar. CloSpan: Mining closed sequential patterns in large datasets. In *Proceedings of 2003 SIAM International Conference on Data Mining (SDM'03)*, 2003.

[Zel02] Andreas Zeller. Isolating cause-effect chains from computer programs. In *FSE*, pages 1–10, New York, NY, USA, 2002. ACM.

[Zel09] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2009.

[ZH02] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, February 2002.