11-5-2018

A Method and Tool for Finding Concurrency Bugs Involving Multiple Variables with Application to Modern Distributed Systems

Zhuo Sun
Florida International University, zsun003@fiu.edu

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

A METHOD AND TOOL FOR FINDING CONCURRENCY BUGS INVOLVING MULTIPLE VARIABLES WITH APPLICATION TO MODERN DISTRIBUTED SYSTEMS

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Zhuo Sun

2018

To: Dean John L. Volakis
College of Engineering and Computing

This dissertation, written by Zhuo Sun, and entitled A Method and Tool for Finding Concurrency Bugs Involving Multiple Variables with Application to Modern Distributed Systems, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Shu-Ching Chen

Peter J. Clarke

Ning Xie

Armando Barreto

Xudong He, Major Professor

Date of Defense: November 5, 2018

The dissertation of Zhuo Sun is approved.

Dean John L. Volakis
College of Engineering and Computing

Andrés G. Gil
Vice President for Research and Economic Development
and Dean of the University Graduate School

Florida International University, 2018

DEDICATION

To my parents, my husband and my daughter.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Xudong He, who offered invaluable advice and patience throughout my Ph.D study at Florida International University. I cannot ask for a better advisor.

I would also like to thank all my committee members for taking their time to serve on the committee and provide feedback for my dissertation work.

ABSTRACT OF THE DISSERTATION

A METHOD AND TOOL FOR FINDING CONCURRENCY BUGS INVOLVING MULTIPLE VARIABLES WITH APPLICATION TO MODERN DISTRIBUTED SYSTEMS

by

Zhuo Sun

Florida International University, 2018

Miami, Florida

Professor Xudong He, Major Professor

Concurrency bugs are extremely hard to detect due to huge interleaving space. They are happening in the real world more often because of the prevalence of multi-threaded programs taking advantage of multi-core hardware, and microservice based distributed systems moving more and more applications to the cloud. As the most common non-deadlock concurrency bugs, atomicity violations are studied in many recent works, however, those methods are applicable only to single-variable atomicity violation, and don't consider the specific challenge in distributed systems that have both pessimistic and optimistic concurrency control.

This dissertation presents a tool using model checking to predict atomicity violation concurrency bugs involving two shared variables or shared resources. We developed a unique method inferring correlation between shared variables in multi-threaded programs and shared resources in microservice based distributed systems, that is based on dynamic analysis and is able to detect the correlation that would be missed by static analysis. For multi-threaded programs, we use a binary instrumentation tool to capture runtime information about shared variables and synchronization events, and for microservice based distributed systems, we use a web proxy to capture HTTP based traffic about API calls and the shared resources they access

including distributed locks. Based on the detected correlation and runtime trace, the tool is powerful and can explore a vast interleaving space of a multi-threaded program or a microservice based distributed system given a small set of captured test runs. It is applicable to large real-world systems and can predict atomicity violations missed by other related works for multi-threaded programs and a couple of previous unknown atomicity violation in real world open source microservice based systems. A limitation is that redundant model checking may be performed if two recorded interleaved traces yield the same partial order model.

TABLE OF CONTENTS

LIST OF TABLES

TABLE PAGE

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

Multi-core hardware is a growing industry trend, for both high performance servers and low power mobile devices. Multi-threaded programs can exploit multi-core processors at their full potential. In the real world, most servers and high-end critical software are multi-threaded. Unfortunately, multi-threaded programs are prone to bugs due to the inherent complexity caused by concurrency. It is difficult to detect concurrency bugs due to the huge number of possible interleavings. Many concurrency bugs escape from testing into software releases and cause some of the most serious computer-related accidents in history, including a blackout leaving tens of millions of people without electricity [1].

Among different types of concurrency bugs, atomicity violation bugs are the most common ones. Atomicity violation bugs are caused by violations to the atomicity of certain code regions without proper synchronization. They widely exist in real world systems and contributed to about 70% of the examined non-deadlock concurrency bugs [2]. Therefore, techniques for detecting atomicity violation bugs are extremely important.

Studies in recent years have been focusing on single-variable atomicity violation. However, those methods are unable to predict or find atomicity violations with multiple variables involved. Many variables are inherently correlated and need to be accessed together with their correlated peers in a consistent manner [3]. These variables need to be either updated together or accessed together to avoid multi-variable atomicity violation.

Beyond the concurrency bugs in multi-threaded systems, there are concurrency bugs in distributed systems. With more and more large-scale distributed systems, there are billions of end users relying on the correctness of the distributed systems. More than 60% of distributed concurrency bugs are triggered by a single untimely message delivery that commits order violation or atomicity violation [4].

1.2 Research Problem

Atomicity is a semantic correctness property for concurrent programs. When proper synchronization is missing to enforce atomicity, atomicity violation bugs may occur, also known as unserializable interleavings that are not equivalent to a serial execution. For the case with a single shared variable involved, whether a two-thread interleaving is equivalent to a serial execution can be determined by checking if its conflict graph is acyclic [5].

Most concurrency bugs involve two threads, instead of a large number of threads, based on the study in [6], in which 101 out of 105 bugs involved only two threads. Thus atomicity violation bugs in a multi-threaded program can be explored through every pair of threads. McPatom [7] is inspired by the works in [2][8], which addressed a special case of unserializable interleavings with three accesses of the same shared variable.

Existing atomicity violation detection tools mostly focus on the bugs that have a single variable involved. The tools study the accesses to the same shared variable. But the atomicity violation bugs caused by unserializable accesses to multiple shared variables actually contribute significantly to the existing known ones [3].

Multiple variable atomicity can be achieved by ensuring the atomicity of each pair of shared variables. So in the sequel, we focus on two-variable atomicity. For

the case with two shared variables involved, a method is desired to check if the two variables are correlated and determine whether a two-thread interleaving is equivalent to a serial execution, that is the problem to discuss in this work.

Predicting atomicity violation for distributed system is also challenging due to its complex non-deterministic nature that involve single resource or multiple resources, and the existence of both pessimistic and optimistic concurrency control.

1.3 Contributions

The typical software development process relies on software testing for quality assurance, but testing cannot ensure every possible scenario is covered. In concurrent systems, it is even more difficult if not impossible to test every feasible thread interleavings due to non-determinism, making concurrency bugs the most troublesome in all types of software bugs. This frustrates both testing and reproduction for bug diagnosis. Many variables are inherently correlated and need to be accessed together with their correlated peers in a consistent manner [3], that makes it even more challenging when the correlated variables are involved in an atomicity violation. Tools to automatically predict multi-variable atomicity violation would save cost in software testing, and even more by avoiding it in production. This dissertation predicts a class of atomicity violation that the existing tools are not able to detect. The contribution of this work is as follows:

1. This dissertation presents a method to infer access correlation from an instrumented interleaved trace that only records events related to atomicity violations. Such an interleaved trace is much smaller than the program behavior in a complete execution. Furthermore, the extracted model and inferred access correlation enable the checking of all alternative traces with the same causal

relationships as the interleaved trace with multiple variables or multiple resources involved.

2. This dissertation proposes a complete set of the patterns of unserializable interleavings involving two threads or processes (most concurrency bugs involve only two threads [6]) containing any number of accesses to multiple shared variable or resources. These patterns generalize and cover the three accesses proposed in [2][8]. These atomicity violation patterns become property specifications to be checked.

3. This dissertation offers a unique prediction tool, for detecting multi-variable atomicity violation bugs through model checking for both multi-threaded programs and microservice based distributed systems.

4. This dissertation reports a couple of previous unknown atomicity violations in real world open source microservice based systems.

1.4 Chapter Organization

The remainder of this dissertation is organized as follows. Chapter 2 discusses the background and how it is related to this work. Chapter 3 presents our work in predicting multi-variable atomicity violations, based on access correlation between variables and atomicity violation pattern of variable accesses. Chapter 4 discusses atomicity violations in microservice based distributed systems and the methods to predict atomicity violations, by applying what was learned from Chapter 3 and studying the difference in distributed systems while there are similarities between protecting shared variables in multi-threaded programs and shared data in distributed systems that run on multiple processes or multiple machines. Chapter 5 concludes the work.

CHAPTER 2

BACKGROUND

This chapter discusses the background related to the research work in this dissertation. Model checking is a method for formally verifying finite-state concurrent systems against the specifications of the systems which are expressed as temporal logic formulas. Efficient symbolic algorithms are developed to traverse the model of the system and check if the specification of the system holds or not.

2.1 Linear Time Temporal Logic

Linear Time Temporal Logic (LTL) formula, proposed in [9], can be used to express both safety and liveness properties, by encoding about the future of paths. Linear Temporal Logic is an infinite sequence of states where each state in time has a unique successor based on a linear time perspective. A system satisfies the LTL formula if and only if the formula holds for all paths of the system. An LTL formula f may contain any lowercase propositional symbol p from a finite set of propositions P, combined with unary or binary, boolean and/or temporal operators, using the grammar shown in the Formula 2.1, Formula 2.2 and Formula 2.3 [10].

$$
\begin{aligned}
f ::= & p \\
& | true \\
& | false \\
& | (f) \\
& | f \ binop \ f \\
& | unop \ f
\end{aligned}
\tag{2.1}
$$

$$
\begin{aligned}
unop ::= & \Box && (always) \\
| & \Diamond && (eventually) \\
| & ! && (logic\ negation)
\end{aligned}
\tag{2.2}
$$

$$
\begin{aligned}
binop ::= & U && (strong\ until) \\
| & X && (next) \\
| & \&\& && (logical\ and) \\
| & \| \| && (logical\ or) \\
| & \Rightarrow && (implication) \\
| & \Leftrightarrow && (equivalence)
\end{aligned}
\tag{2.3}
$$

Figure 2.1 illustrates the intuition behind the main LTL operators in the Formula 2.1. An interpretation for a LTL formula is an infinite word $\xi = x_0 x_1 ...$ over the alphabet 2^P, i.e. a mapping from the naturals to 2^P [11]. The elements of 2^P are interpreted as assigning truth values to the elements of P: elements in the set are assigned true, elements not in the set are assigned false. We write ξ_i for the suffix of ξ starting at x_i. The semantics of LTL is then shown in the following.

1. $\xi \models q$ iff $q \in x_0$, for $q \in P$, that means q holds at the current state.
2. $\xi \models \neg\varphi$ iff not $\xi \models \varphi$, that means φ doesn't hold at the current state.
3. $\xi \models \varphi \wedge \psi$ iff $\xi \models \varphi$ and $\xi \models \psi$, that means both φ and ψ hold at the current state.
4. $\xi \models \varphi \vee \psi$ iff $\xi \models \varphi$ or $\xi \models \psi$, that means either φ or ψ hold at the current state.

5. $\xi \models X\varphi$ iff $\xi_1 \models \varphi$, that means φ holds at the next state.

6. $\xi \models \varphi \cup \psi$ iff there is an $i \geq 0$ such that $\xi_i \models \psi$ and $\xi_j \models \varphi$ for all $0 \leq j < i$, that means φ holds at least until ψ which holds at the current or a future state.

7. $\xi \models \Box\varphi$ iff $\xi_i \models \varphi$ for all $i \geq 0$, that means φ holds at the entire subsequent path,

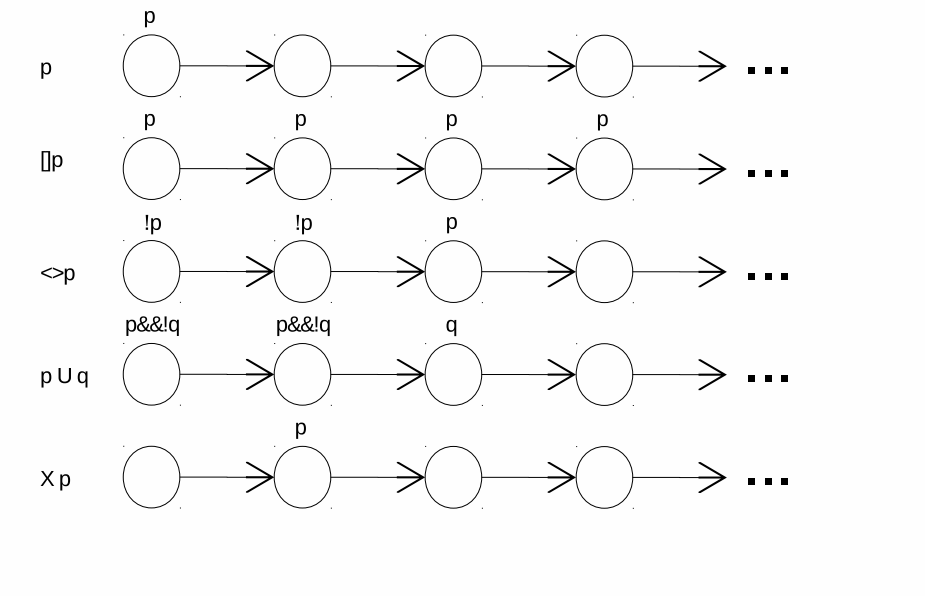
8. $\xi \models \Diamond\varphi$ iff there is an $i \geq 0$ such that $\xi_i \models \varphi$, that means φ eventually holds at a state in the subsequent path,

LTL formulas can specify both safety and liveness, where safety means something "bad" will never happen and liveness means something "good" will happen. For example, no violation of mutual exclusion is a safety property, can be specified as $\Box(\neg inCS_A \vee \neg inCS_B)$ for a pair of threads A and B where $inCS$ means in critical sections. And starvation freedom is an example of livenss, whenever process A wants to enter the critical section, provided process B never stays in the critical section forever, A gets to enter eventually. Starvation freedom can be specified as $\Box\Diamond(B = InCS \Rightarrow X(B = OutCS)) \Rightarrow \Box(A = RequestCS \Rightarrow \Diamond(A = InCS))$.

2.2 Model Checking

Almost all computing systems involve asynchronous concurrency, in the form of threads or message passing. The design and analysis of concurrent systems has proved to be one of the most vexing practical problems in computer science [12]. The difficulty is largely caused by the "interleavings" problem. That is, the developer of a concurrent system faces with the huge number of possible orderings of actions that can be generated by independent processes or threads, which causes errors

Figure 2.1: Intuition for the main LTL operators



that are hard to reproduce. The errors might not manifest frequently but they make systems unacceptably unreliable. Model checking is one possible solution to the problem.

Edmund M. Clarke defines model checking as follows in the book [13]. Let AP be a set of atomic propositions. A Kripke structure M over AP is a four tuple $M = (S, S_0, R, L)$ where

1. S is a finite set of states.
2. $S_0 \subseteq S$ is the set of initial states, and can be omitted when we are not concerned with the set of initial states S_0.
3. $R \subseteq S \times S$ is a transition relation that must be total, that is, for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s')$.
4. $L : S \to 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

Given a Kripke structure $M = (S, R, L)$ that represents a finite-state concurrent system and a temporal logic formula f expressing some desired specification, the model checking problem is to find the set of all states in S that satisfy f: $\{s \in S \mid M, s \models f\}$

Ken McMillan defined the model checking in the foreword of [12]:"Model checking is a fully automated verification technique that constructs a graph representing all possible states of the system and the transitions between them. This state graph can be thought of as a finite folding of an infinite computation tree containing all possible executions of the system. Using the state graph, we can definitively answer questions about the system's behavior posed in temporal logic, a specialized notation for specifying systems that evolve in time."

Unfortunately, because the "computation tree" explicitly represents all possible interleavings of concurrent executions, the size of the state graph is huge even for simple systems, that is the well known state explosion problem [14]. Partial order methods have been proposed and developed to address the state explosion problems in many research works, such as [15][16] and unfolding approaches [17] [12]. What's more, [18] combines the partial order methods with on-the-fly model checking [19, 20].

In this dissertation based on traces of accesses of shared variable or resources, although the total number of possible interleavings to check can explode quickly as the number of accesses increase, however, the number of actual interleavings are drastically smaller due to the constraints imposed by happen-before relationships between threads. Another way to vastly reduce the possible interleavings is to reduce number of variables to check, in this dissertation we take advantage of the nature of atomicity violations and considers only a pair of threads or processes and accesses to a single or two shared variables or resources at one time, groups all reading

event sequences in each thread or process into atomic blocks to achieve partial order reductions.

2.3 Spin Model Checker

Spin model checker [10] uses the combination of partial order methods with on-the-fly model checking [18] to reduce the number of reachable states that must be explored to complete a verification.

SPIN takes design specifications written in the verification language Promela (a Process Meta Language) [21] as models to check, and it takes correctness claims specified in the syntax of standard Linear Temporal Logic (LTL) [9] as the properties to verify. The specification of a concurrent system in PROMELA consists of one or more user-defined process templates, or proctype definitions, and at least one process instantiation, in which the process can represent threads in multi-threaded programs or machines in distributed systems. The process templates define the behavior of processes.

In Promela programs, (1) the execution of every statement is conditional on its executability. Statements are either executable or blocked. The executability is the basic means of synchronization. A process can wait for an event to happen by waiting for a statement to become executable. A condition can only be executed (passed) when it holds. If the condition does not hold, execution blocks until it does. (2) Variables are used to store either global information about the system as a whole, or information local to one specific process, depending on where the declaration for the variable is placed. Variables can be declared as arrays, for example, *short locked*$[N]$ declares an array of N short. (3) The state of a variable can only be changed or inspected by processes. The behavior of a process is defined in a proc-

type declaration. (4) A proctype definition only declares process behavior, it does not execute it. Initially just one process will be executed: a process of type init, that must be declared explicitly in every Promela specification, to instantiate processes. (5) By prefixing a sequence of statements enclosed in curly braces with the keyword *atomic* it is indicated that the sequence is to be executed as one indivisible unit, non-interleaved with any other processes. (6) The selection structure using keyword if and fi contains multiple execution sequences, each preceded by a double colon. Only one sequence from the list will be executed. A sequence can be selected only if its first statement is executable. The first statement is therefore called a guard. If more than one guard is executable, one of the corresponding sequences is selected nondeterministically. If all guards are unexecutable the process will block until at least one of them can be selected.

In this dissertation we automatically generate Promela code for all synchronization primitives [7]. We present Promela code for mutex locks as an example. We model synchronization events to capture the happen-before relationships between threads, to prune infeasible interleavings. The Promela code shown in Figure 2.2 models the POSIX Thread routines *pthread_ mutex_ lock* and *pthread_ mutex_ unlock*. The atomic construct groups indivisible statements together to ensure no interleaving within an atomic sequence. *Lock* inline function accepts a lock l as its argument. If lock l is not locked, *Lock* function locks it and sets the owner to the thread that is the predefined variable $\_pid$ for the executing process in Promela. If lock l is in locked status, no guards are executable so that the thread is blocked until lock l is available according to Promela semantics. *Unlock* inline function simply sets lock l to unlocked status. It is exactly what is required to model locking and unlocking of a mutex lock. Figure 2.3 gives an example of proctype that is used to simulate a thread or a process.

Figure 2.2: Promela Code Modeling Mutex Locks

```
#define NUM_LOCKS 100
short locked[NUM_LOCKS] = -1;
inline Lock(l) {
 if
 ::atomic{(locked[l] == -1) -> locked[l] = _pid}
 fi;
}
inline Unlock(l) {
 assert(locked[l] == _pid);
 locked[l] = -1;
}
```

Figure 2.3: A Sample of Partial Promela Code

```
proctype t1()
{
  ...
}
proctype t2()
{
  ...
}
init
{
  run t1();
  run t2();
  ...
}
```

SPIN essentially is a model checker generator. It takes models in PROMELA and properties in LTL as inputs to generate a model checker, the generated model checker search through the states to give counter examples for the LTL properties. The counter examples can be used to drive simulation and presentation, in the context of this dissertation, of predicted atomicity violation as shown in Chapter 3 and Chapter 4.

2.4 High Level Petri Nets

Petri nets, developed in early 1960s by Carl Adam Petri, are a data driven formal model for modeling the control structure and dependency of concurrent and distributed systems [22]. A Petri net is defined by a net structure $N = (P, T, F)$, where P is a finite set of places represented by circles, T is a finite set of transitions represented by bars or boxes and F is the set of directed arcs $F \subseteq P \times T \cup T \times P$. Places represent system states; transitions represent state transitions; and directed arcs represent the control flows and dependencies between states.

High level Petri nets (HLPNs) generalize the original Petri nets with data definition and processing capabilities: (1) each place needs to have a data type from a domain of $Types$, (2) each arc needs to have a label from a domain of $Labels$, (3) each transition needs to have a logic formula from a domain of $Formulas$ defining the precondition and post-condition of a transition firing, and (4) each place is empty or contain some initial tokens from a domain of $Tokens$ as initial states. The above static semantic domains are traditionally defined using an algebraic specification $\Sigma = (S, Op, Eq)$ with a family of sorts (types) S and the associated operations Op, and a set of equations Eq defining the meaning of the operations. In the context of this dissertation we use a simplified definition of a type of HLPNs called predicate

transition (PrT) nets [23]. The static semantics of a PrT net is defined using the following mappings:

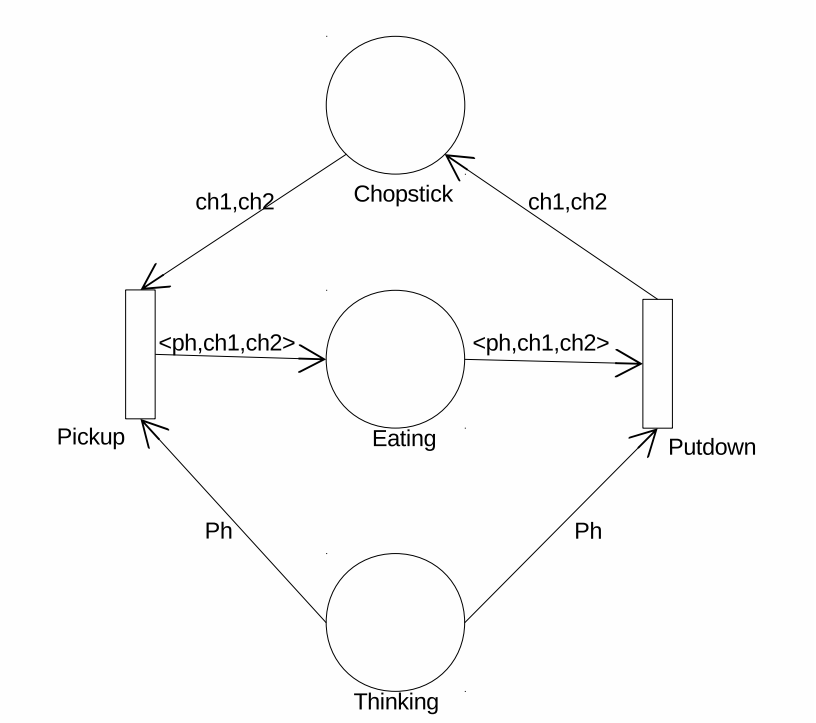
1. $\varphi : P \rightarrow Types$ associates each place p in P with a type in $Types$. In a PrT net, places are often called predicates to highlight their roles as in predicate logic.
2. $L : F \rightarrow Labels$ is a sort-respecting labeling of arcs. We use the following abbreviation in the sequel: $\bar{L}(x, y) = L(x, y)\ if\ (x, y) \in F, or\ \emptyset\ otherwise$;
3. $R : T \rightarrow Formulas$ is a well-defined constraining mapping, which associates each transition t in T with a first order logic formula defining the meaning of the transition.
4. $M_0 : P \rightarrow Tokens$ is a sort-respecting initial marking which assigns a set of tokens to each place p in P.

Based on the above definitions, a PrT net is defined as $PN = (P, T, F, \Sigma, \varphi, L, R, M_0)$. A PrT net is executable and its dynamic semantics is defined on markings and transition firings. Markings of a PrT net PN are mappings $M : P \rightarrow Tokens$ and denotes the states of PN. Given a marking M, a transition $t \in T$ is enabled if there is a token substitution α of variables on the incoming arcs and in the constraint formula satisfying: $\forall p : p \in P.(\bar{L}(p, t) : \alpha) \subseteq M(p)) \wedge R(t) : \alpha$. An enabled transition t under marking M with substitution α can fire and results in a new marking M' defined by $M'(p) = M(p) - \bar{L}(p, t) : \alpha$ for $p \in P$, which is denoted as $M[t/\alpha > M'$. As in low level Petri nets, two enabled transitions may fire at the same time as long as they are not in conflict by firing one transition disable another transition. An execution sequence, $E = M_0[t_0/\alpha_0 > M_1[t_1/\alpha_1 > ... > M_n[t_n/\alpha_n > ...$, of PN starts from the initial marking and contains successive execution steps T_i of non-conflict transition firings. The behavior of PN is the set of all E.

Taking dining philosophers problem as an example, Figure 2.4 shows a PrT net specification. Two philosopher states are denoted by places *Thinking* and *Eating* respectively, two philosopher actions are denoted by two transitions *Pickup* and *Putdown*, and the available chopstick state are defined by the place *Chopstick*. The PrT net definition is as follows: (1) Place Types: $\varphi(Thinking) = PHIL$, $\varphi(Eating) = PHIL \times CHOP \times CHOP$, $\varphi(Chopstick) = CHOP$, where types $PHIL$ and $CHOP$ are induced from integers. (2) Arc Labels: $L(Thinking, Putdown) = ph$, and the rest are similar as shown in Figure 2.4. (3) Transition Constraints: $R(Pickup) = (ch1 = ph) \wedge (ch2 = ph \oplus 1)$, $R(Putdown) = true$ where $\oplus$ is modulus k addition. (4) Initial Marking: $M_0 = \{m_k | k = 2, 3, ...\}$ where $m_k(Thinking) = \{1, 2, ...k\}$ and $m_k(Eating) = \emptyset$ and $m_k(Chopstick) = \{1, 2, ..., k\}$.

Since Petri net is very well suited to model the message passing, shared data and distributed locks that are all we need to analyze atomicity violation in distributed systems, this disseration automatically build Petri net models for predicted atomicity violations in distributed systems to help manually confirming them being a false positive or not.

Figure 2.4: Dining Philosophers Problem in PrT nets



CHAPTER 3

PREDICTING MULTI-VARIABLE ATOMICITY VIOLATION

3.1 Introduction

Multi-core hardware is a growing industry trend, for both high performance servers and low power mobile devices. Multi-threaded programs can exploit multi-core processors at their full potential. In the real world, most servers and high-end critical software are multi-threaded. Unfortunately, multi-threaded programs are prone to bugs due to the inherent complexity caused by concurrency. It is difficult to detect concurrency bugs due to the huge number of possible interleavings. Many concurrency bugs escape from testing into software releases and cause some of the most serious computer-related accidents in history, including a blackout leaving tens of millions of people without electricity [1].

Among different types of concurrency bugs, atomicity violation bugs are the most common ones. Atomicity violation bugs are caused by violations to the atomicity of certain code regions without proper synchronization. They widely exist in the real world systems and contributed to about 70% of the examined non-deadlock concurrency bugs according to a research in the year 2006 [2]. Therefore, techniques for detecting atomicity violation bugs are extremely important.

The studies in recent years have been focused on single-variable atomicity violation. However, those methods are unable to predict or find atomicity violations with multiple variables involved. Many variables are inherently correlated and need to be accessed together with their correlated peers in a consistent manner [3]. These variables need to be either updated together or accessed together to avoid two-variable atomicity violation.

This chapter presents a method for predicting two-variable atomicity violation, based on access correlation between variables and atomicity violation pattern of variable accesses and in this chapter we make the following contributions [24]:

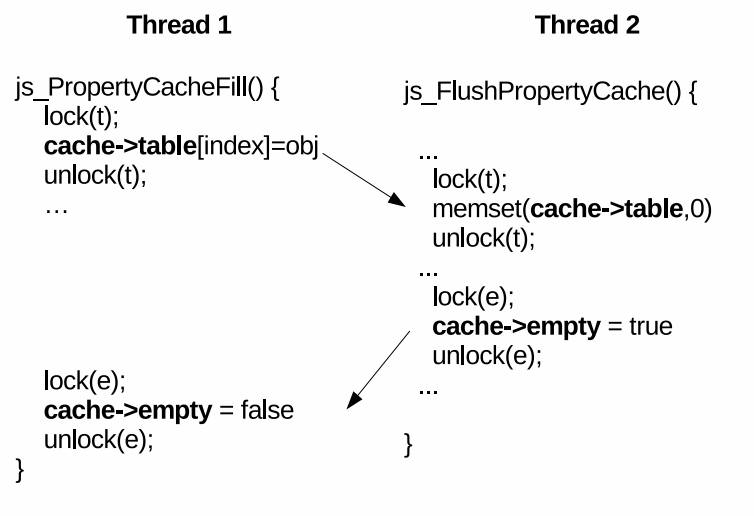
1. A method to infer access correlation from an instrumented interleaved trace that only records events related to atomicity violations. Such an interleaved trace is much smaller than the program behavior in a complete execution. Furthermore the extracted thread model and inferred access correlation enable the checking of all alternative traces with the same causal relationships as the interleaved trace with multiple variables involved.
2. A complete set of the patterns of unserializable interleavings involving two threads (most concurrency bugs involve only two threads [6]) containing any number of accesses to multiple shared variable (either user-defined or every word sized dynamically allocated memory accessed by multiple threads). These patterns generalize and cover the three accesses proposed in [2][8]. These atomicity violation patterns become property specifications to be checked.
3. A unique prediction tool - McPatom-MV, for detecting two-variable atomicity violation bugs through model checking.

3.2 Motivation

Multiple variable atomicity can be achieved by ensuring the atomicity of each pair of shared variables. So in the sequel, we focus on two-variable atomicity.

Existing atomicity violation detection tools mostly focus on the bugs that have a single variable involved. The tools study the accesses to the same shared variable. But the atomicity violation bugs caused by unserializable accesses to multiple shared variables actually contribute significantly to the existing known ones [3]. There are

Figure 3.1: Mozilla bug 1

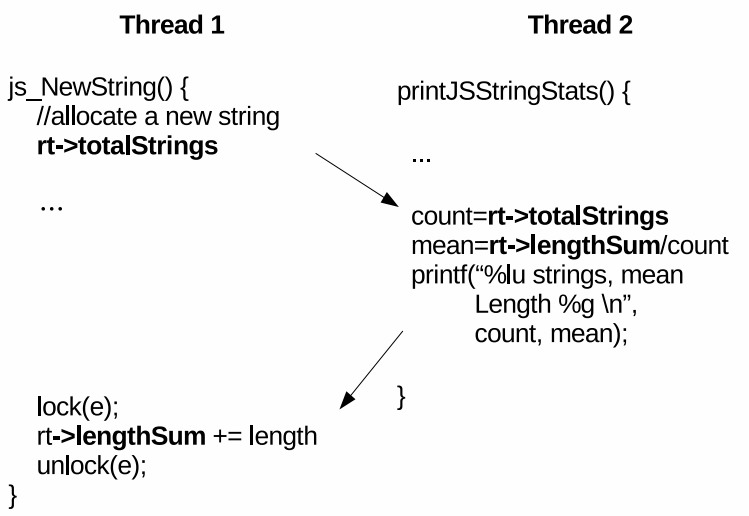


a few well known two-variable atomicity violations. In the sequel, we use real-world examples [3] to show what could be atomicity violation bugs in the real world, and also one of the most challenging ones to show the benefit of our methods against the existing tools.

Figure 3.1 shows an example from Mozilla-0.8, each single shared variable is synchronized properly by using the lock so that there is no data race and no single variable atomicity violation. However, the two variables *table* and *empty* as part of the same structure *cache* are correlated by its nature, *empty* is used to indicate whether *table* is empty or not so that the two variables have to be updated together. In the interleaved execution as shown in Figure 3.1, they are not updated together though, which ends up with *table* being empty but with the variable *empty* being false that indicates *table* is not empty.

Figure 3.2 gives an example from Mozilla-0.9, the two variables *totalStrings* and *lengthSum* are part of the same structure *rt*, and are correlated to each other. The interleaved execution as shown in the figure is a violation of atomicity in terms

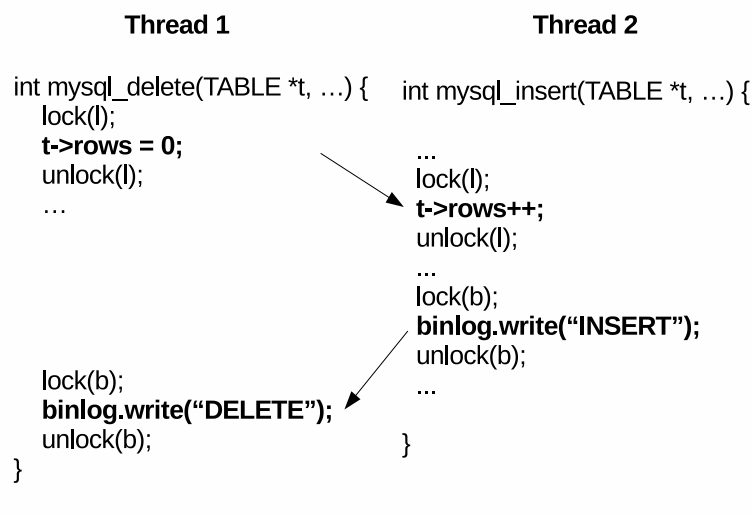
Figure 3.2: Mozilla bug 2



of two variables because it reads an intermediate value and results in inconsistent values between the variables.

In the examples discussed above, the correlated variables are accessed in adjacent statements, however, with complicated cases, the statements can be encapsulated into different functions so that the statements are not adjacent to each other any more but their access to the correlated variables will be next or very close to each other in the sequence of statements during runtime. Static source code analysis methods can identify such correlation across functions by analyzing the function call graph, but is unable to guarantee to find all such cases due to its limitation of static analysis. Analyzing the correlation based on dynamic runtime traces makes it easier to analyze such correlation because it does not have to analyze the call graph. It can find such correlation as long as it manifests in the traced execution. It has an assumption though, the correlated variables need to be accessed in the sample runnings which should be able to cover most cases with good suits of test cases. The

Figure 3.3: MySQL bug



runtime method can complement static methods and has the benefit that it won't miss the common cases.

Figure 3.3 shows an example from MySQL that separate our method from other similar methods in the related works. It may cause failure of database recovery - table t has one row as in the interleaving in the figure but it will have no row during database recovery due to the sequence of "INSERT" then "DELETE" in database log.

Our method based on dynamic execution traces is able to detect it as discussed in the following.

3.3 Predicting Single Variable Atomicity Violation

In this section, we briefly discuss the tool McPatom as in [7], which is generalized in this paper to support two-variable atomicity violation.

3.3.1 Description of the Partial Order Thread Model

A multi-threaded program runs with multiple threads and variables. The access to local variables has no impact to concurrency, thus not be able to cause concurrency bugs. The variables allocated dynamically in the heap can be potentially accessed by multiple threads, can be involved in concurrency bugs. Those variables can be potentially involved in concurrency bugs are defined as shared variables, that are addresses of global variables and every word sized dynamically allocated memory accessed by multiple threads. The same memory address does not necessarily mean the same variable though since it can be reused through the memory management functions, so memory allocation and deallocation instructions are also monitored in order to differentiate the variables in case the same memory address is reused.

For a multi-threaded program P, an execution $\sigma = s_1, ..., s_n$ of is a sequence of executed statements. An execution can be projected to a sequence of annotated shared variable accesses and synchronization events, which is the trace to analyze in this work. Formally, a trace, $\tau = e_1, ..., e_m$ is a sequence of events where each event $e_i (1 \leq i \leq m)$ is a tuple $\langle tid_i, timestamp_i, action_i \rangle$ in which tid_i is a thread handle, $timestamp_i$ is a time stamp based on real time and $action_i$ is one of the following: (read/write/allocate/deallocate, a shared variable), (a synchronization routine, a synchronization variable) or (a thread management operation, a thread handle). McPatom uses POSIX Threads in which a synchronization routine is a routine related to semaphores, mutex locks, condition variables and barriers, does not handle user-defined synchronization primitives. McPatom also assumes a shared variable as a synchronization variable if it is accessed by synchronization routines, thus does not treat its accesses as shared variable accesses.

Given a trace $\tau = e_1, ..., e_m$ containing shared variable accesses and synchronization events, a partial order thread model $(E_\tau, \prec)$ is defined as follows:

1. $E_\tau = \{e_i \mid e_i \; in \; \tau\}$

   (a) $\prec$ is a partial order relation such that, for any $e_i, e_j \in E$ $(i \neq j)$, $e_i \prec e_j$ iff

      i. $tid_i = tid_j$ and $i < j$, or

      ii. $tid_i \neq tid_j$, $action_i = (Signal, cvar)$, $action_j = (Wait, cvar)$ and $\forall k \centerdot ((j < k < i) \wedge (action_k \neq (Signal, cvar))$ in which $cvar$ is a condition variable, or

      iii. $tid_i \neq tid_j$, $action_i = (Wait, bvar)$ and $(i < j) \wedge \exists k \centerdot ((tid_k = tid_j) \wedge (k < j) \wedge action_k = (Wait, bvar) \wedge \forall h \centerdot ((tid_h = tid_k) \Rightarrow \neg(k < h < j)))$ in which $bvar$ is a barrier variable, or

      iv. $tid_i \neq tid_j$, $action_i = (Create, tid_j)$, or

      v. $tid_i \neq tid_j$, $action_j = (Join, tid_i)$.

   (b) Mutual exclusion: for any $e_i, e_j, e_m, e_n \in E$ $(i \neq j \neq m \neq n)$, $e_j \prec e_m$ or $e_n \prec e_i$ iff

      i. $tid_i = tid_j$, $action_i = (Lock, lvar)$, $action_j = (Unlock, lvar)$, and

      ii. $tid_m = tid_n$, $action_m = (Lock, lvar)$, $action_n = (Unlock, lvar)$.

The partial order above defines the causal relation and is similar to the happened-before relation given in [25]. The above definition ensures (1) shared variable accesses within the same thread are ordered, and (2) the constraint of synchronization is preserved regarding its impact to the shared variable accesses across multiple

threads. Therefore, it captures alternative traces that obey the same causal relation as τ and thus equivalent to the original trace, and each alternative trace τ' is a result of rearranging the order of some shared variable accesses across different threads without breaking the constraint by $\prec$. The partial order thread model enables exploration of all possible alternative traces that correspond to a set of feasible interleavings in a multi-threaded program. However, the model provides an over-approximation without considering data-flow, thus cannot guarantee each alternative trace captured in the model can be projected back to some feasible interleaved execution in the multi-threaded program P, that is the reason of false positives.

3.3.2 Implementation of the Partial Order Thread Model

McPatom uses Pin binary instrumentation framework [26] to instrument a running executable and capture runtime information into a trace, specifically including, every access (read/write/allocate/deallocate) to every shared variable and every synchronization event using POSIX Thread (locks, condition variables, barriers, thread joining and etc.). For each event in the trace, McPatom also finds the corresponding source code information including file name and line number through the debug information contained in the executable, that can be used to help to locate the predicted bugs and also to find the variable correlation across multiple traces. A sample of a partial trace is shown in Figure 3.4.

3.3.3 Three-access Atomicity Violation

Many recent works focused on three-access atomicity violations [2][8][6], which involve one shared variable, two threads and three accesses to the variable. If two

```
4913812332, 1515882591, sample.c-812, Read, threads
4913812332, 1515882591, sample.c-126, Create, 4915489248
4915489248, 1515882591, sample.c-310, Lock, lockVar
4915489248, 1515882591, sample.c-311, Read, sharedVar
4915489248, 1515882591, sample.c-311, Write, sharedVar
4915489248, 1515882591, sample.c-312, Signal, condVar
4915489248, 1515882591, sample.c-313, Unlock, lockVar
```

Figure 3.4: A Sample of a Partial Trace (The format of each line: thread handle, unix epoch timestamp, file name - line number, action)

consecutive accesses of a shared variable in a thread are interleaved with an access to the same variable from another thread, and the interleaving is unserializable, the atomicity of the two consecutive accesses is violated and it is a potential atomicity violation bug. The explanation of unserializable interleavings of three accesses and many real world atomicity violation bugs can be found in [2]. The related works above focus on three-access atomicity violations because (1) there are many real world atomicity violation bugs involving only three accesses, and (2) checking only three accesses in a pair of threads can greatly reduce the complexity of algorithms.

Definition 1 (Serializability)**.** A two-thread interleaving is serializable if and only if it is equivalent to a serial execution, which executes a code region without another thread interleaved in between. The code region is typically enforced as atomic explicitly in the code.

Conflict graph is used in the context of concurrency control in databases [5]. It contains 1) a node for each memory access; 2) an arc from node A_i to node A_j if A_i precedes and conflicts with A_j where conflicts means at least one of A_i and A_j is a write.

Definition 2 (Serializability with single variable)**.** A single-variable two-thread interleaving is equivalent to a serial execution if and only if its conflict graph is acyclic according to the Serializability Theorem [5].

Figure 3.5: Conflict graph for a single-variable two-thread interleaving

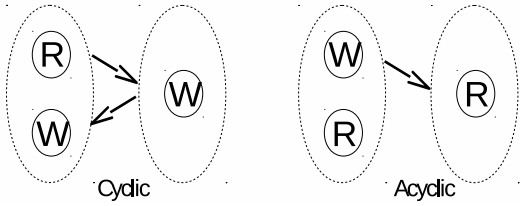


Figure 3.6: Unserializable Interleavings with two threads. In (1)(2)(3)(5), W in Thread 2 unexpectedly changes the value; In (4), An intermediate value in Thread 1 is read by Thread 2.

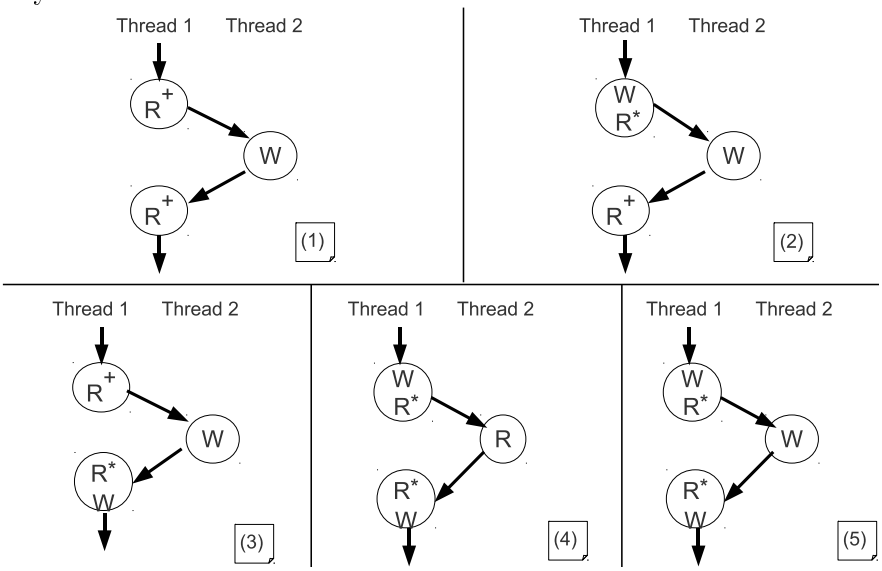


Figure 3.5 shows two examples of conflict graphs for a single-variable two-thread interleaving, one is cyclic and another is acyclic between two threads. The cyclic one is unserializable while the acyclic one is serializable.

Figure 3.6 shows all possible scenarios of unserializable interleavings with only one access from Thread 2. If any of the unserializable interleaving patterns are matched, it indicates a potential atomicity violation. In the figure, R^+ means one or more read accesses and R^* means zero or more read accesses.

3.4 Variable Correlation Analysis

The discussions in Section 3.2 provides the motivation to predict two-variable atomicity violation, and focus on a pair of variables, however, only the variables that are correlated with each other can potentially cause atomicity violations. Many variables are inherently correlated and need to be accessed together with their correlated peers in a consistent manner. A pair of variables need to be accessed together, that is the atomicity to ensure otherwise there could be atomicity violation.

The correlation is usually in the developer's mind or even not realized by developers especially for developers maintaining the software or components but not the original author, so the most important thing to predict atomicity violation bugs with two-variable involved is to infer the correlation automatically, since it is very impracticable to expect developers to enforce or somehow mark the correlation between variables, if not impossible.

Definition 3 (Variable Correlation)**.** Two variables x and y are correlated if every time variable x is accessed, variable y is also accessed shortly afterward, formally denoted as $access(x) \Rightarrow access(y)$.

The important question to answer is how we measure the correlation. There can be multiple possible ways.

It can be measured in source code distance [3]: if two accesses appear in the same function with less than $MaxDistance$ statements apart, these two accesses are considered together, where $MaxDistance$ is an adjustable threshold. The limitation is it assumes all correlated accesses happen in the same function, however, the correlated accesses can happen in different functions that are called in the same function. The measure in source code distance is not able to cover that problem.

It can also be measured in dynamic execution distance, that is the distance in the trace of dynamic execution. The measure is excluded in [3] because it is believed that two correlated accesses can easily be separated by a loop or a function invocation and thus have a large dynamic execution trace. We design the trace to be just for memory accesses of shared variables, that makes dynamic execution distance a good measure, and avoid the limitation in the measure of source code distance discussed above.

3.5 Algorithm to Infer Access Correlation from a Single Trace

The trace is a sequence of memory accesses, the idea is to infer access correlation from their distance in the trace, which indicates the possibility whether any pair of accesses should be atomic.

We instrument the memory allocation/deallocation instructions to get memory addresses that should be treated as shared variables. When a memory address is deallocated and allocated again, we treat it as a separate variable.

3.5.1 Memory Access Correlation Table

For n memory addresses or variables $v_1 ... v_n$, the memory access table is a $n \times n$ matrix C, $C = [c_{xy}]$ where $1 \leq x, y \leq n$ and c_{xy} is calculated as follows for the correlation between v_x and v_y. For memory accesses $A_i(v_x)$ and $A_j(v_y)$ in the trace, where $A_i(v_x)$ is the ith access that read or write the variable v_x and $A_j(v_y)$ is the jth access that read or write the variable v_y, if there is a sequence $A_i A_k ... A_m A_j$, suppose the memory accesses from A_k to A_m access variables other than v_x and v_y and $j \geq i + 1$, add the distance $j - i - 1$ to a list of distances d_{xy}. After getting

distances of all pairs of v_x and v_y added into the list d_{xy}, we can calculate c_{xy} based on d_{xy}.

In the list of distances d_{xy}, there could be shorter ones and longer ones in which there could be a presence of outliers. One of the common ways of finding outliers in a list is to mark any point that is more than two standard deviations from the mean as a potential outlier. But the presence of outliers could have a strong effect on the mean and the standard deviation, making those ways unreliable to find outliers. Median Absolute Deviation (MAD) [27] proposes to use absolute deviation around the median as a way of dealing with the problem of outliers. We use MAD to filter out outliers in d_{xy} then get the mean of the rest in d_{xy} as c_{xy}, as shown in Algorithm 3.1, where 1.4826 is a constant linked to the assumption of normality of the data, disregarding the abnormality induced by outliers.

3.5.2 Recommendation of Possible Access Correlation

The lower value in the table $C = [c_{xy}]$ above, the more likely the pair of access is correlated. It is hard to define a threshold to decide whether a pair of accesses should be treated as correlated, but it is easy to give a sorted list for either prioritized checking atomicity violation or manual confirmation.

3.6 Algorithm to Infer Access Correlation from Multiple Traces

To infer access correlation from multiple traces, we need to identify the same variables across multiple traces. There are two types of shared variables: 1) global variables; 2) variables dynamically allocated in the heap.

Program 3.1 Quantify the correlation between pairs of shared variables

```
FindCorrelatedVars(traceFile)
{
  Find pairsInAtLeastTwoThreads;
  for (x,y) in pairsInAtLeastTwoThreads
  {
    for each thread thd
    {
      AccessList ax = accesses of x;
      AccessList ay = accesses of y;
      AccessList a = sorted(ax + ay);
      i = 0;
      while (i + 1 < a.length)
      {
        if (variable of a[i] !=
            variable of a[i+1]):
          DistanceList dxy;
          dxy.append(a[i+1]-a[i]);
        i += 1
      }
    }
    mad = median(abs(dxy-median(dxy)))
          * 1.4826;
    for d in dxy
    {
      AbsoluteDeviationList ad;
      ad[d] = abs(d-median(dxy)) / mad;
      if ad[d] < 2:
        distanceListNoOutlier.append(d);
    }
    AverageDistance c[x,y] =
        average(distanceListNoOutlier);
  }
  return sorted(c)
}
```

3.6.1 Global Variables

With the symbol table contained in the executable, we are able to find the mapping from address to variable name.

3.6.2 Variables Dynamically Allocated in the Heap

With debug information built into the executable, we can find the file name and line numbers in the source code for each read or write access. Because our goal is to find the correlation between shared variables, we can assume the shared variables accessed from the same line of code are highly correlated, thus we can assume there is only one shared variable for any line of source code.

With the assumption above, we can treat a unique pair of file name and line number accessing shared memory as a shared variable so that we can infer access correlation across multiple traces.

3.7 Serializability of Two-Variable Two-Thread Interleavings

The definition of serializability of single-variable atomicity violations in the sections above is not applicable to the atomicity violation with multiple variables involved. Two-variable atomicity can be achieved by ensuring the atomicity of each pair of shared variables. So in the sequel, we focus on two-variable atomicity violation. Given two shared variables x and y, and $A \in \{Read, Write\}$, $R = Read$, $W = Write$, Let R_x denote reading variable x, W_x denote writing variable x, R_y denote reading variable y and W_y denote writing variable y. When treating each pair of correlated shared variables as a single one, Definition 2 and the patterns in McPatom [7] can be applied and resulting is called as McPatom-MV1. MUVI [3] requires the

writes of both variables in forming atomicity violation. Both of the above methods can find possible atomicity violation, however, are overall restrictive and can result in false positive predictions.

Definition 4 (Conflict graph with two variables)**.** Given two variables x and y, there is at least one access of x and y in one thread denoted as A_x^1 and A_y^1, and at least one access of x and y in another thread denoted as A_x^2 and A_y^2, the conflict graph contains a node for each memory access, and an arc if A_v^1 conflicts with A_v^2 i.e. at least one of them is a write.

The definition of conflict graph above is generalization of the conflict graph for single variable [28] [5].

Definition 5 (Serializability with two variables)**.** A two-variable two-thread interleaving is equivalent to a serial execution if the two variables are correlated and the conflict graph is acyclic between two threads.

3.8 Predict Two-Variable Atomicity Violation

This section discusses the simple method using the existing McPatom with patterns of single variable atomicity violation, and another method extending McPatom with patterns of two variables atomicity violation. Two methods are independent of each other.

The framework contains the following major steps: (1) using Pin [26] to instrument an interleaved execution of a multi-threaded program and to record an interleaved trace containing only atomicity violation impacting events including all shared variable accesses and all synchronization routines (locks, condition variables, barriers and thread management events); (2) projecting the interleaved trace into a

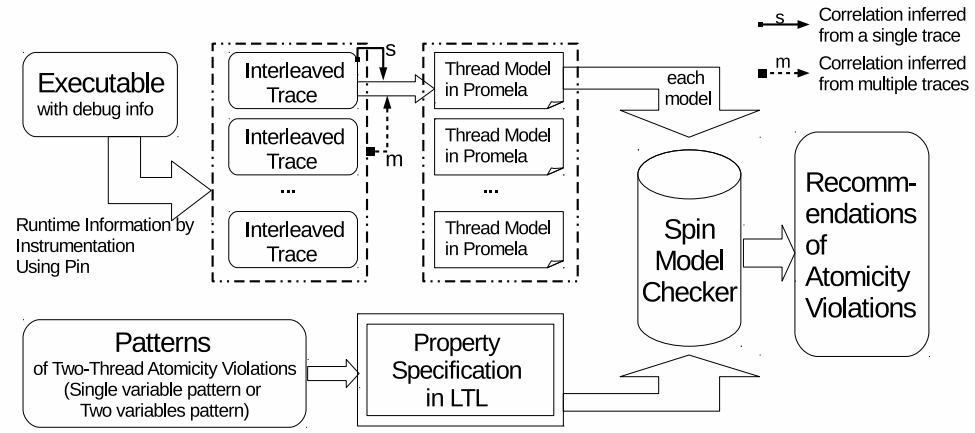
partial order thread model of abstract threads, which maintains the causal relation within actual threads imposed by the synchronization routines, and treats two correlated shared variable as a single one using the patterns of single variable or keeps two correlated shared variables using the patterns of two variables; (3) automatically translating the partial order thread model into a Promela program for model checking in Spin [10]; (4) defining a complete set of atomicity violation patterns as in Figure 3.6 [7] involving a pair of threads accessing every single shared variable and automatically translating them into temporal logic formulas; (5) defining a complete set of atomicity violation patterns involving a pair of threads accessing every pair of two shared variables and automatically translating them into temporal logic formulas; (6) using Spin to model check the atomicity violation patterns; and (7) mapping the violation reported in Spin to the execution trace in the original multi-threaded program.

Figure 3.7 gives an overview of McPatom framework. If using the patterns of two-thread atomicity violations with a single variable, the thread model in Promela treats the two correlated shared variables as a single one. If using the patterns of two-thread atomicity violations with two variables, the thread model in Promela keeps the two correlated shared variable.

3.8.1 McPatom-MV1: Use Existing McPatom with Patterns of Single Variable Atomicity Violation

Using the pairs of correlated memory accesses inferred above, we can integrate McPatom to find atomicity violations for each individual trace, by treating each pair of correlation memory accesses as a single variable, since McPatom works on predicting single variable atomicity violation from a single trace.

Figure 3.7: Overview of the method predicting atomicity violations

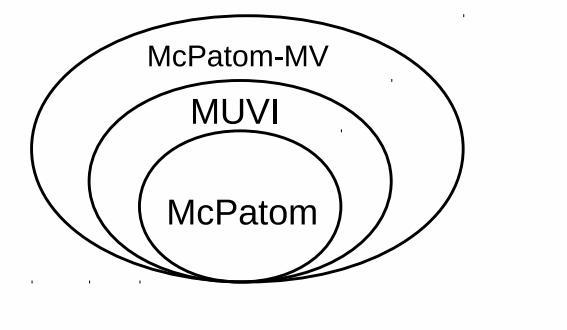


When analyzing a single trace, multiple executions of the same line of source code can access different memory addresses that are dynamically allocated in the heap, and different memory addresses are counted as different variables. Let L_1, L_2 be two lines of code denoted by file name and line number, we can infer access correlation between L_1 and L_2 as discussed above. Let's denote a line in the trace as t, there are two lines of traces t_1 and t_2 and two memory addresses A_1 and A_2, such that $t_1 = L_1A_1$ and $t_2 = L_2A_2$ which means t_1 accesses the memory address A_1 from the line of source code L_1 and the similar for t_2.

If A_1 and A_2 are correlated as the algorithm in Section 3.5, treat t_1 and t_2 as the same shared variable.

If L_1 and L_2 are correlated as the algorithm in Section 3.6, treat t_1 and t_2 as the same shared variable.

Figure 3.8: Comparison of methods about coverage of atomicity violation



3.8.2 McPatom-MV2: Extend McPatom with Patterns of Two Variables Atomicity Violation

McPatom-MV1 above is straightforward to leverage the existing McPatom. However, it can report more atomicity violations than other methods as shown in Figure 3.8 that potentially means more false positives. To reduce false positives, we propose the following access patterns specifically for two-variable atomicity violation. The result is called McPatom-MV2, having better coverage than MUVI [3] because MUVI only consider the inconsistent updates that start with write and it cannot take all shared variables into consideration due to the limitation of static analysis.

3.8.2.1 Patterns of Two-thread Atomicity Violations involving Two Variables

In the sequel, a two-variable atomicity violation refers to a two-thread atomicity violation involving any number of accesses of two shared variable x and y, and $A \in \{Read, Write\}$, $R = Read$, $W = Write$, R_x denotes reading variable x, W_x denotes writing variable x, R_y denotes reading variable y and W_y denotes writing variable y. This section gives a set of patterns covering all possible two-variable atomicity violations.

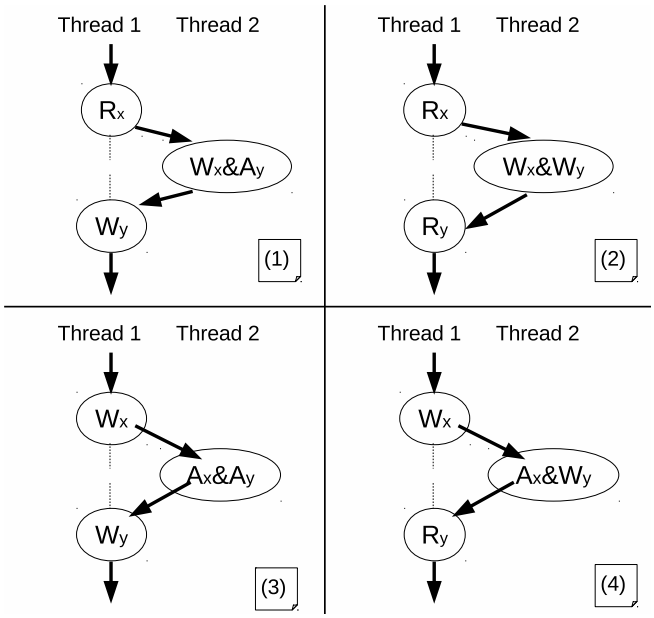


Figure 3.9: Unserializable Interleavings with two variables and two threads.

Based on Definition 4 and Definition 5, we propose the pattern as in Figure 3.9 as a complete set of patterns for unserializable interleaving that can be used to predict atomicity violations with two variables and two threads involved. In pattern 1, R_x and W_y are interleaved by W_xA_y or A_yW_x from thread 2 where W_x from thread 2 unexpectedly changes x that makes it unserializable if W_y is dependent on R_x in thread 1; in pattern 2, R_x and R_y are interleaved by W_xW_y or W_yW_x in thread 2 where W_x and W_y unexpectedly change x and y that makes R_x and R_y in thread 1 reading inconsistent value; in pattern 3, W_x and W_y are interleaved by A_xA_y or A_yA_x in thread 2 where thread 2 could read inconsistent value of x and y or write x to make x and y inconsistent; in pattern 4, W_x and R_y are interleaved by W_yA_x or A_xW_y in thread 2 where W_y from thread 2 unexpectedly change y and A_x could unexpectedly read or change x from W_x of thread 1 thus causing unexpected value in R_y.

Theorem 1 (Completeness of the set of Patterns in Figure 3.9)**.** *The set of patterns in Figure 3.9 is complete, i.e. it includes all possible unserializable interleavings between two threads with two variables involved.*

Proof. Let $A_1^{t_1}, A_2^{t_2}, ..., A_n^{t_n}$ be a sequence of atomic accesses in an interleaved execution of two threads with two variables involved, in which $A_i^{t_i}$ ($t_i \in \{1,2\}$, $A_i^{t_i} \in \{Read, Write\}$, $1 \leq i \leq n$) denotes an atomic access from thread t_i to the two shared variables. Let every subsequence of $A_1^{t_1}, A_2^{t_2}, ..., A_n^{t_n}$ be of the form 1) $X_1^1, X_2^2, Y_3^2, Y_4^1$ where X_1^1 and Y_4^1 of Thread 1 are accesses $A_i^{t_i}(t_i = 1)$, X_2^2 and Y_3^2 of Thread 2 are accesses $A_i^{t_i}(t_i = 2)$, or of the similar form 2) $X_1^1, Y_2^2, X_3^2, Y_4^1$, 3) $X_1^1, X_2^2, Y_3^1, Y_4^2$, or 4) $X_1^1, Y_2^2, Y_3^1, X_4^2$. The forms are shown in Figure 3.10. Form 2 can be proved similarly, Form 3 is impossible to have conflict graph as there is no cycle between two threads and Form 4 can be reduced to Form 2. The following proof is based on Form 1. Let P_i be pattern i. If $X_1^1, X_2^2, Y_3^2, Y_4^1$ does not match with any of the patterns in Figure 3.9, $X_1^1, X_2^2, Y_3^2, Y_4^1$ satisfies $\neg P_1 \wedge \neg P_2 \wedge \neg P_3 \wedge \neg P_4$. Since operator $\wedge$ is commutative, we can select a specific order and carry out an incremental analysis of possible $X_1^1, X_2^2, Y_3^2, Y_4^1$ based on each of $P_i(1 \leq i \leq 4)$. The details of each step are omitted and as a result, when $X_1^1, X_2^2, Y_3^2, Y_4^1$ satisfies $\neg P_1 \wedge \neg P_2 \wedge \neg P_3 \wedge \neg P_4$, $X_1^1, X_2^2, Y_3^2, Y_4^1$ can only be one of the following:

1) $X_1^1 = W_x$, $X_2^2 = R_x$, $Y_3^2 = R_y$, $Y_4^1 = R_y$

2)$X_1^1 = W_x$, $X_2^2 = W_x$, $Y_3^2 = R_y$, $Y_4^1 = R_y$

3) $X_1^1 = R_x$, $X_2^2 = R_x$, $Y_3^2 = R_y$, $Y_4^1 = R_y$

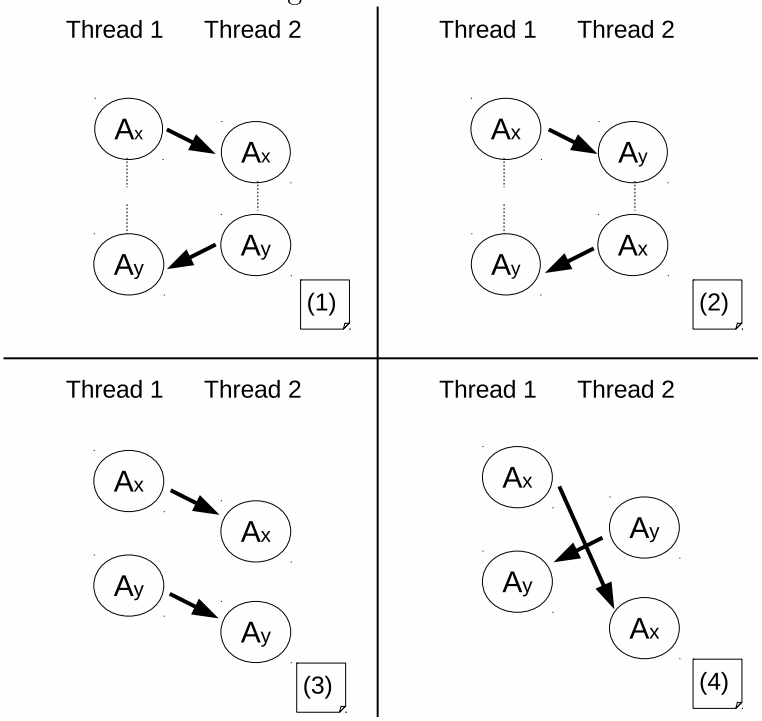
4) $X_1^1 = R_x$, $X_2^2 = W_x$, $Y_3^2 = R_y$, $Y_4^1 = R_y$

5) $X_1^1 = R_x$, $X_2^2 = R_x$, $Y_3^2 = W_y$, $Y_4^1 = R_y$

6) $X_1^1 = R_x$, $X_2^2 = R_x$, $Y_3^2 = W_y$, $Y_4^1 = W_y$

7) $X_1^1 = R_x$, $X_2^2 = R_x$, $Y_3^2 = R_y$, $Y_4^1 = W_y$

Figure 3.10: All Interleaving Forms of Two Variables and Two Threads



According to the Serializability Definition 5, an interleaved sequence is serializable if and only if its conflict graph is acyclic. All of the above seven patterns are serializable. Therefore, the completeness of the set of patterns in Figure 3.9 is proved.

□

3.8.2.2 Automatically Encoding Traces to Promela Code

McPatom-MV2 automatically encodes a trace to multiple Promela files with each one containing a pair of shared variables. It defines each shared variable v in the pair of shared variables (x, y) as a *short* in Promela, automatically assigns a unique value for all reading accesses and a unique value for all writing accesses in each thread. Formally, let $rw \in \{r, w\}$, and *tid* be thread ID, then v=rw+tid for each access of

v. McPatom sets r to be *0*, and w to be *10000*. For example, given two threads: *t1(tid=1)* and *t2(tid=2)*, and a shared variable *v,* McPatom makes assignments as below for each scenario of accesses.

- Assign *10000+1* to v for each writing access of v in thread *t1*,
- Assign *1* to v for each reading access of v in thread *t1,*
- Assign *10000+2* to v for each writing access of v in thread $t2$,
- Assign *2* to v for each reading access of v in thread *t2.*

McPatom-MV2 automatically generates Promela code for all synchronization primitives, like McPatom [7]. Figure 3.11 gives a sample of partial Promela code encoding a trace.

3.8.2.3 Automatically Encoding Atomicity Violation Patterns into Linear Time Temporal Logic (LTL) Formulas

A pattern is a sequence of accesses, for example, $R_y^1 W_y^2$. Every two adjacent accesses in the sequence can be captured in a LTL formula, for example $R_y^1 W_y^2$ can be captured in $y == r+1$ && $X(y == r+1\ U\ y == w+2)$ where X denotes *Next* and "U" denotes *Until*. For every pair of shared variable x and y, and every pair of threads $t1$ and $t2$, McPatom-MV2 automatically defines a pair of LTL formulas including Formula 3.1 for each pattern in Figure 3.9 and another LTL formula reversing the view of t_1 and t_2, another pair of LTL formulas reversing the view of x and y. Let x and y be a pair of shared variables, $r = 0$ and $w = 10000$ as defined in 3.8.2.2, $A_i \in \{r, w\}$, and $tid_i \in \{1, 2\}$ where $i \in \{1, 2, 3, 4\}$ as there are four accesses in the pattern.

```
proctype t1() { ... }
proctype t2()
{
  Lock(lock1);
  v_tab = 0 + 2;     /* mysql-binlog.c - 53 - (In trace:
     15612) */
  v_tab = 10000 + 2;     /* mysql-binlog.c - 53 - (In
     trace: 15613) */
  Unlock(lock1);     /* mysql-binlog.c - 54 - (In trace:
     15617) */
  Lock(lock2);
  v_numLines_binlog = 0 + 2;     /* mysql-binlog.c - 32 -
     (In trace: 95478) */
  v_numLines_binlog = 10000 + 2;     /* mysql-binlog.c -
     32 - (In trace: 95479) */
  Unlock(lock2);     /* mysql-binlog.c - 60 - (In trace:
     95482) */

  ThdJoin!2;
}

init
{
  run t1();     /* mysql-binlog.c - 103 - (In trace: 4) */
  run t2();     /* mysql-binlog.c - 104 - (In trace: 5) */
  run t3();     /* mysql-binlog.c - 106 - (In trace: 6) */
  ...
}
```

Figure 3.11: A Sample of Partial Promela Code

$$
\begin{aligned}
&[]! <> ((x == A_1 + tid_1) \&\& \ X \\
&\quad ((x == A_1 + tid_1) U \\
&\quad\quad ((x == r + tid_2) U \\
&\quad\quad\quad (x == A_2 + tid_2 \ \&\& \ X \\
&\quad\quad\quad\quad (x == A_2 + tid_2 \ U \\
&\quad\quad\quad\quad\quad ((y == r + tid_3 \ \&\& \ x == A_2 + tid_2) \ U \\
&\quad\quad\quad\quad\quad\quad ((y == A_3 + tid_3 \ \&\& \ x == A_2 + tid_2) \ \&\& \ X \\
&\quad\quad\quad\quad\quad\quad\quad ((y == A_3 + tid_3 \ \&\& \ x == A_2 + tid_2) U \\
&\quad\quad\quad\quad\quad\quad\quad\quad ((y == r + tid_4 \ \&\& \ x == A_2 + tid_2) U \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad (y == A_4 + tid_4 \ \&\& \ x == A_2 + tid_2)))))))))\}
\end{aligned}
\tag{3.1}
$$

where "[]" denotes *Always*, "!" denotes *Logical Negation*, "<>" denotes *Eventually*. These formulas specify that the atomicity violation patterns do not occur. Note that the patterns need to be extended to cover all forms in Figure 3.10, and the LTL formula allows extra read accesses of the same variable in the same thread to precede each of the four accesses in the pattern, that is r in the Formula 3.1, because it happens in the real world and it preserves the conflict graph in Definition 4. Formula 3.1 captures a pattern with $A_1A_2A_3A_4$ where A_1 and A_2 are accesses of x and A_3 and A_4 are accesses of y, and each of $A_2A_3A_4$ can be preceded with insignificant read accesses of the same variable from the same thread.

Using Figure 3.9 (2) as a concrete example, the pattern $[]! <> R_x^1 W_x^2 W_y^2 R_y^1$ can be captured in the formula in LTL below.

$$
\begin{aligned}
&[]! <> ((x == r+1) \&\& X \\
&((x == r+1) U \\
&((x == r+2) U \\
&(x == w+2 \ \&\& \ X \\
&(x == w+2 \ U \\
&((y == r+2 \ \&\& \ x == w+2) \ U \\
&((y == w+2 \ \&\& \ x == w+2) \ \&\& \ X \\
&((y == w+2 \ \&\& \ x == w+2) U \\
&((y == r+1 \ \&\& \ x == w+2) U \\
&(y == r+1 \ \&\& \ x == w+2)))))))))\}
\end{aligned}
\tag{3.2}
$$

In Formula 3.2, Line 1 and 2 denote A_1 of x from thread 1, that is R_x^1; Line 3 denotes insignificant accesses r of x from thread 2 that exist or doesn't exist in the trace; Line 4 and Line 5 denote A_2 of x from thread 2, that is W_x^2; Line 6 denotes insignificant accesses r of y while making sure x is still $w+2$; Line 7 and Line 8 denote A_3 of y from thread 2, that is W_y^2; Line 9 denotes insignificant accesses r of y; Line 10 denotes A_4 of y from thread 2, that is R_y^1. Therefore, Formula 3.2 captures $[]! <> R_x^1 W_x^2 W_y^2 R_y^1$ and ensures that pattern $R_x^1 W_x^2 W_y^2 R_y^1$ in Figure 3.9 (2) does not occur in the partial order thread model. LTL formula to capture $[]! <> R_x^1 W_y^2 W_x^2 R_y^1$ is similar.

3.9 Evaluation

All of our experiments are conducted on a machine with 2 Core 2.3GHz CPU, 4GB of memory, running Debian 8.7 as the operating system. We use Pin tool 2.14 for

binary instrumentation and SPIN tool 6.4.6 for model checking. The tool and the results can be downloaded from https://users.cs.fiu.edu/~zsun003/qrs18/

3.9.1 Variable Correlation Analysis

Our variable correlation analysis based on dynamic trace is efficient. For MUVI [3] it takes about 3 hours to infer variable correlation for 3-4 million lines of code. We applied our method to the latest Apache Httpd 2.4.29 and it takes about 3 seconds to infer variable correlation for 1 million lines of code. We inferred 971 pairs of variable correlation, comparable to the one reported in MUVI. We not only give the list of variable correlation for reference by programmers or other tools but also give the qualified weight for each pair in terms of average distance between the pair of variables. MUVI took a sample of 100 correlations and manually whether they are true to give a false positive rate. We believe it is an error-prone process and it would be better for application developers to justify it so that we provide quantified weight and a ranked list.

3.9.2 Two-Variable Atomicity Violation Detection

Table 3.1 shows four real-world two-variable atomicity violation bugs. Our method can detect all of them. For MySQL-169, the correlation between $t \rightarrow rows$ and *binlog* is conditional, [3] is unable to detect it because $t \rightarrow rows$ can be accessed many times and *binlog* is only modified after the final update of $t \rightarrow rows$, and the static method doesn't get the correlation. ColorSafe [29] is unable to detect it because the t and *binlog* are not allocated together. UNICORN [30] is likely unable to detect it because the length between accesses of t and *binlog* is beyond the limit of its sliding window size.

Table 3.1: Atomicity violation bugs with multiple variables involved

BugId	App	Description
Moz-js1	Mozilla-Suite v0.9	Writes of correlated variables are interleaved by remote thread's writes, causing the empty flag to be false for an empty table and system crash. Shown in Figure 3.1
Moz-js2	Mozilla-Suite v0.8	Writes of correlated variables are interleaved by remote thread's reads which read intermediate inconsistent values. Shown in Figure 3.2
MySQL-2011	MySQL v4.0.16	http://bugs.mysql.com/bug.php?id=2011 Read to log file's name and log file are interleaved by remote thread rotating logs thus writing log file's name and log file.
MySQL-169	MySQL v3.23.56	http://bugs.mysql.com/bug.php?id=169 Table deletion and log writing are interleaved by remote thread's table insertion and log writing. Shown in Figure 3.3

Regarding performance, it takes about average 2 minutes for Spin model checkers to check all properties for a pair of variables.

Using MySQL-169 as an example, here is how to run the tool that generates a trail file for each possible atomicity violation, and how the trail can tell what is the interleaved execution with related source code information that is an atomicity violation. Running the tool is easy, for example,

```
~/McPatomMV$ sh runMcPatomMV.sh testdata/mysql-binlog12/
    trace.out Benchmark/mysql-binlog
```

where the first parameter specifies the path to store the trace output which can be any empty folder, and the second parameter specifies the executable to run and trace. After the tool is finish, it prints the results. But to check results anytime, the script as shown in Figure 3.13 can be used to print the trail again. Each trail is an atomicity violation. Note that 10000+1 means Write from thread 1, 0+2 means

Figure 3.12: Atomicity Violation in MySQL-169

```
== Atomicity violation in the bug MySQL-169
  Thread 1                    Thread 2
Write(v_tab)
                            Write(v_tab)
                            Write(v_numLines_binlog)
Write(v_numLines_binlog)
```

Read from thread 2, so the p23.trail in Figure 3.13 below is a pattern that violates atomicity (Pattern 3 in the paper).

3.10 Related Works

There are many recent works on tackling atomicity violations. Some works proposed techniques to detect atomicity violations on actual program executions through testing [31], runtime monitoring ([2], [28], and [32]) or predict atomicity violations based on actual program executions [7][33][8][34][35][6][36][37][38][39][40][41].

We discuss the works related to two-variable atomicity violation in this section.

3.10.1 MUVI

MUVI [3] automatically infers commonly existing two-variable access correlations through code analysis. It combines static program analysis and data mining techniques to automatically infer two-variable correlations. Firstly, it parses the source code and collects each function's variable access information, including the set of variables accessed within each function, the access types and locations in source code. Secondly, it uses a frequent pattern mining technique to find out all the variable sets that frequently appear together and produces a pool of variable access correlation candidates.

Figure 3.13: Examples of Experiment Result

```
== The tool to show the atomicity violation and related
   source code.
~/McPatomMV/testdata/mysql-binlog12$ ../../printtrail.sh
spin_numLines_binlog-tab.pml.p23.trail
  v_tab = 0 + 3;     /* mysql-binlog.c - line 85 - (In trace:
     13) */
  v_tab = 10000 + 3;     /* mysql-binlog.c - line 89 - (In
     trace: 230) */
  v_tab = 10000 + 1;     /* mysql-binlog.c - line 40 - (In
     trace: 334) */
  v_tab = 0 + 2;     /* mysql-binlog.c - line 53 - (In trace:
     8) */
  v_tab = 10000 + 2;     /* mysql-binlog.c - line 53 - (In
     trace: 9) */
  v_numLines_binlog = 0 + 2;     /* mysql-binlog.c - line 32
     - (In trace: 28) */
  v_numLines_binlog = 10000 + 2;     /* mysql-binlog.c - line
     32 - (In trace: 41) */
  v_numLines_binlog = 0 + 1;     /* mysql-binlog.c - line 32
     - (In trace: 349) */
  v_numLines_binlog = 10000 + 1;     /* mysql-binlog.c - line
     32 - (In trace: 350) */
spin_numLines_binlog-tab.pml.p24.trail
  v_tab = 0 + 3;     /* mysql-binlog.c - line 85 - (In trace:
     13) */
  v_tab = 10000 + 3;     /* mysql-binlog.c - line 89 - (In
     trace: 230) */
  v_tab = 0 + 2;     /* mysql-binlog.c - line 53 - (In trace:
     8) */
  v_tab = 10000 + 2;     /* mysql-binlog.c - line 53 - (In
     trace: 9) */
  v_tab = 10000 + 1;     /* mysql-binlog.c - line 40 - (In
     trace: 334) */
  v_numLines_binlog = 0 + 1;     /* mysql-binlog.c - line 32
     - (In trace: 349) */
  v_numLines_binlog = 10000 + 1;     /* mysql-binlog.c - line
     32 - (In trace: 350) */
  v_numLines_binlog = 0 + 2;     /* mysql-binlog.c - line 32
     - (In trace: 28) */
  v_numLines_binlog = 10000 + 2;     /* mysql-binlog.c - line
     32 - (In trace: 41) */
```

MUVI considers two types of variables: global variables and structure fields. It cannot take all shared variables into consideration, that is an inherent limitation for the static method because it is difficult to find shared variables precisely in a static way and it is impossible to deal with all variables so that it has to choose a limited set of variables for consideration. Instead, the dynamic method used in our works can find correlations even if they are dynamically allocated and not belong to the same structure. In another aspect, for the correlated structure fields that are not shared, finding them is useless to find atomicity violations.

3.10.2 Generation of Unit Tests for Correlated Variables

In [42] test cases are automatically generated to prevent the race condition in correlated variables in concurrent programs. Its approach to identifying correlations between variables is based on static analysis of given program, similar to MUVI [3]. Besides identifying the variables that are accessed often near to each other, it also considers variables that are data and/or control dependent on each other. The variable written in an assignment is considered to be data-dependent on each variable that is read in the assignment. The variable written in a control flow branch is considered to be control-dependent on variables that are read in the branching condition. The approach as an extension of MUVI [3] shares the same limitation that it cannot take all shared variables into consideration due to it static analysis. Our method based on dynamic analysis can catch those dependencies thus be able to find the same correlation.

3.10.3 ColorSafe

ColorSafe [29] groups related data into colors, and then monitors access interleavings in the "color space". It has two modes of operation: debugging mode and deployment mode. In debugging mode, it detects only interleavings that are actually unserializable, i.e., it cannot predict bugs that do not manifest. In deployment mode, it attempts to dynamically avoid atomicity violations by detecting when an atomicity violation is likely to happen and dynamically starting a special form of transaction to prevent an unserializable interleaving from happening. However, since it does not take synchronizations into consideration, it produces false positives and in correspondence triggers unnecessary bug avoidance actions.

It is related to our work in detecting two-variable atomicity violation based on related data. It explores both manual coloring and automatic coloring. For automatic coloring, it is based on memory allocation, that is, it gives the same color to data allocated together. However, there are a lot of examples on two-variable atomicity violation in which the correlated variables are not allocated together. [3] gives two examples in which although the fields in each pair belong to the same structure, i.e. they are allocated together, but they do not have access correlation as they are accessed together in only 3–4 functions and are accessed separately in 68 or 87 functions. Instead, our work infers correlation from execution traces using heuristics, in order to detect bugs regardless how the correlated variables are allocated.

3.10.4 UNICORN

UNICORN [30] detects order violations, single-variable, and two-variable atomicity violations. It monitors pairs of memory accesses for each shared variable, combines

the pairs into problematic patterns, and ranks the patterns by their suspiciousness scores. There are three steps. It firstly executes a program multiple times, monitors memory-access pairs within a fixed-sized sliding window, and marks each program execution as either passing or failing. Secondly, it combines memory-access pairs into problematic memory-access patterns using a second fixed-sized sliding window for maintaining pairs. Thirdly, it computes the suspiciousness of the patterns and orders them in decreasing order of suspiciousness to recommend possible bugs.

The pairs include only the directly adjacent accesses for each shared variable. And, for single variable atomicity violation, it follows the three-accesses pattern; for multiple variable atomicity violation, it follows the four-accesses pattern. Thus UNICORN has a limitation not be able to check atomicity violation involving more accesses.

It uses a small fixed-sized window to identify the most suspicious memory-access patterns, to reduce time and space overhead. Hence for those accesses with a distance more than the length of the window, it is impossible to detect.

It instruments the source program statically using LLVM, and monitors shared variable accesses during runtime. It does not instrument the synchronization instructions and it is based on observed executions, therefore, it cannot predict concurrency bugs and can only detect concurrency bugs when they manifest. In the experiment, to increase the probability of program failures, it inserted random artificial delays into the programs.

3.11 Summary

Concurrency bugs are extremely hard to detect using testing techniques due to huge interleaving space. As the most common non-deadlock concurrency bugs, atomicity

violations are studied in many recent works, however, those methods are applicable only to single-variable atomicity violation. This chapter presents enhanced tools McPatom-MV1 and McPatom-MV2 based on McPatom using model checking to predict atomicity violation concurrency bugs involving two variables. We developed a unique method inferring the correlation between variables, that is based on dynamic analysis and is able to detect the correlation that would be missed by static analysis.

The tools McPatom-MV1 and McPatom-MV2 is powerful and can explore a vast interleaving space of a multi-threaded program based on a small set of instrumented test runs. It is applicable to large real-world systems and can predict atomicity violations missed by other related works. A limitation inherent from McPatom is that redundant model checking may be performed if two recorded interleaved traces yield the same partial order thread model.

CHAPTER 4

ATOMICITY VIOLATION IN DISTRIBUTED SYSTEMS

4.1 Introduction

Reliability of distributed systems is extremely important, especially with the prevalence of big data and cloud computing, that bring a lot of stakeholders such as application's developers and customers into the world of distributed systems and their inherent complexity. Unfortunately, distributed concurrency bugs exist everywhere, are very challenging to detect during testing, and similar to the concurrency bugs for multi-threaded programs. Various models have been proposed [43] to analyze or predict the reliability of large-scale distributed systems, but reliability issues with these systems still exist. Distributed system's reliability depends on the reliability of each individual component, what's more, it also depends on the reliability of the communication between components that typically runs over network and by nature is not reliable, therefore, untimely message delivery or lost messages are common and need to be tolerated to ensure the reliability of the whole distributed systems. Transparent fault detection and fault recovery scheme are typically implemented to provide seamless interaction to end users, that includes methods of automatic redelivery of messages and thus causes duplicated delivery of messages. All those factors above contribute to the complexity of ensuring the reliability of distributed systems.

More than 60% of distributed concurrency bugs are triggered by a single untimely message delivery that commits order violation or atomicity violation [4, 44]. We study the multiple variable atomicity violations for multi-threaded programs in Chapter 3. This chapter aims to extend what we learn from Chapter 3 and develop methods to analyze and predict atomicity violation in distributed systems, by

studying the shared data and distributed lock being used to protect shared data in distributed systems.

Studies in the recent years have been focused on the infrastructure of distributed systems that manage the hardware and coordination between them. However, even the infrastructure is reliable, the application that is running on top of the infrastructure is not guaranteed to be reliable, actually with the prevalence of applications moving to microservices architecture that is essentially distributed systems to provide demanded scalability and availability, it is becoming common to find the need to improve the reliability of applications running on distributed infrastructure. Unfortunately distributed concurrency bugs for applications don't receive the same attention as for the underlying infrastructure, developers lack awareness of distributed concurrency bugs and don't have tools to assist debugging. As a result, distributed concurrency bugs are common and developers tend to live with the bugs until they cause serious problems because they are rare and extremely hard to find.

4.2 Motivation

Developers assume atomicity or transactions in distributed systems, in a similar way as multi-threading programs. In distributed systems, there is a layer processing data, also known as data layer or transaction layer, and there is another layer as application layer that orchestrates transactions onto the transaction layer.

The transaction layer could be implemented using traditional relations database that provides ACID guarantee (ACID is atomicity, consistency, isolation, and durability, more discussions are shown in Section 4.3.1), in that case, atomicity violation can still happen across different transactions, similar to the way atomicity violation

can manifest in a data race free multi-threading program that uses locks but not enough locks.

The transaction layer could also be implemented using key-value store that doesn't provide ACID guarantee, in that case, atomicity violation is more likely to happen, and it is the application layer to do transactions in a way that can prevent atomicity violation. That is what to be discussed in this section.

As shown in Program 4.1, there are two functions that are two possible transactions without distributed locks running on two separated machines. Let the account balance starts with 50,000, the first transaction withdraws 25,000 that changes the balance to be 25,000, and the second transaction deposits 5,000 that changes the balance to be 30,000. However, if two customer representatives access the same bank account simultaneously, data integrity could be violated because of atomicity violation, in the following order.

1. Customer representative 1 fetches bank Account with balance as 50,000
2. Customer representative 2 fetches bank Account with balance as 50,000
3. Customer representative 1 withdraws 25,000 and updates bank Account balance to be 25,000
4. Customer representative 2 deposits 5,000 and updates Bank Account balance = 55,000

The above case is a critical disaster to the bank, can be fixed by using a distributed lock to protect read and write (that is Get and Update in the example).

In a practical case, a customer not only has a balance but also have other customer information such as addresses, contact information, and transactions which can be stored into many different tables. As a practice to enable system scaling horizontally, the bank doesn't use foreign keys in any of those tables so that each

Program 4.1 Atomicity Violation Without Distributed Locks

```
withdraw (accountId, withdrawAmount) { // Bank withdrawal
    transaction
        // Fetch BankAccount object from key-value data
           store
        BankAccount account = datastore.Get(accountId) as
            BankAccount;
        // assume balance = 50,000 and withdrawAmount =
           25,000

        if (account != null && account.IsActive)
        {
            // Withdraw money and reduce the balance
            account.Balance -= withdrawAmount;

            // Update key-value store with new balance =
               25,000
            datastore.Update(accountId, account);
        }
}

deposit (accountId, depositAmount) { // Bank deposit
   transaction
        // Fetch BankAccount object from key-value data
           store
        BankAccount account = datastore.Get(accountId) as
            BankAccount;
        // assume balance = 25,000 and depositAmount =
           5,000

        if (account != null && account.IsActive)
        {
            // Deposit money and increment the balance
            account.Balance += depositAmount;

            // Update cache with new balance = 30,000
            datastore.Update("Key", account);
        }
}
```

table is isolated from each other. To ensure consistency it makes a lot of sense to lock a customer during an update, at customer level or finer-grained level such as customer address level, to avoid atomicity violation.

4.3 Background - Data Consistency and Data Access in Distributed Systems

Microservices is a software architecture pattern to application development in distributed systems being adopted by large companies for cloud infrastructure and applications in the cloud, such as Amazon AWS and Microsoft Azure. That has grown in popularity in recent years. It is a type of software architecture where large applications are made up of small, self-contained components working together through APIs as the interface between components. Each service has a limited scope, concentrates on a particular task and is highly independent. This setup allows easy and large scaling individual components whenever necessary, making the overall systems essentially distributed systems, where components located on networked computers communicate and coordinate their actions by passing messages.

How the microservices communicate with each other depends on application's requirements, but many developers use RESTful API over HTTP with JSON format for the data, and naturally to facilitate object-oriented programming the data is typically stored in document-oriented database, or document store, that is designed for storing, retrieving and managing JSON formatted documents and are one of the main categories of NoSQL databases.

Document-oriented databases are inherently a subclass of the key-value store, another NoSQL database concept. Document databases contrast strongly with the traditional relational database (RDB). Relational databases generally store data

in separate tables that are defined by the programmer, and a single object may be spread across several tables. Document databases store all information for a given object in a single instance in the database, and every stored object can be different from every other. This makes mapping objects into the database a simple task, normally eliminating anything similar to an object-relational mapping. This makes document stores attractive for programming web applications, which are subject to continual change in place, and where the speed of deployment is an important issue. One of the advantages frequently cited for document-oriented databases is their performance. Operating with simpler data structures than those of SQL databases, document-oriented databases have often shown faster speeds of storage and retrieval. However, while they may offer advantages in handling larger volumes of unstructured data more rapidly, they typically lack the ACID (atomicity, consistency, isolation, and durability) properties because of trading off ACID compliance for other properties, such as 100% availability and faster speeds.

In this section, we discuss ACID of traditional relational databases and what that helps to ensure correctness of distributed systems and discuss CAP theorem which applies to all distributed systems especially for key-value stores that don't provide the ACID guarantee. Based on CAP theorem, we list different consistent types of a single item in distributed systems, in which eventual consistency is popular to provide better availability than strong consistency but strong consistency is sometimes required but ignored by developers and once it is ignored in many different places of the distributed systems and it becomes very hard to find them before they cause serious harms. Our goal essentially is to analyze and predict such violation of strong consistency that leads to atomicity violation bugs. We also discuss the data access pattern commonly being used in recent distributed systems, that present us opportunities to analyze distributed systems without having access to source code

by examining the data access in the network traffic. Finally, we discuss distributed locks and write-with-version that can be used to achieve strong consistency when necessary to avoid atomicity violation bugs.

4.3.1 ACID of Traditional Relational Database

The ACID properties of traditional relational database help to ensure the data integrity. While it is typically not present for key-value stores, it is helpful to understand what we need to ensure correctness of a distributed system. ACID consists of 4 features.

- Atomic: The transaction should either succeed to a new state or fail to the original state. In other words, all or nothing should be committed.
- Consistent: Any transaction will bring the system from one valid state to another." Note that it is different from the consistency in CAP Theorem, as the consistency of ACID is for the state of the whole system, however, the CAP is about the consistency of a single item.
- Isolated: Transactions cannot interfere with each other. This feature ensures only one transaction can occur simultaneously for a single item.
- Durable: Once a transaction has been committed, it will remain so. The database should persist everything after the transaction is completed.

4.3.2 CAP Theorem

The CAP theorem [45, 46], also named Brewer's theorem after computer scientist Eric Brewer, states that it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees:

- Consistency: Every read receives the most recent write or an error
- Availability: Every request receives a non-error response - without the guarantee that it contains the most recent write
- Partition tolerance: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

In particular, the CAP theorem implies that in the presence of a network partition that is part of any distributed system, one has to choose between consistency and availability. Note that consistency, as defined in the CAP theorem, is quite different from the consistency guaranteed in ACID database transactions. The consistency in ACID of traditional databases is for the whole system while the consistency in CAP theorem is for a single item. We discuss the consistency of a single item in the following section to clarify why atomicity violation bugs exist in distributed systems.

4.3.3 Consistency Types

To discuss further on the consistency of a single item, there are a few different consistency types.

- Strong consistency: Strong consistency is a consistency model where all subsequent accesses after the update to a single item will always return the updated value.
- Weak consistency: It is a consistency model used in distributed computing where subsequent accesses cannot guarantee returning the updated value.
- Eventual consistency: Eventual consistency is a special type of weak consistency model in which if no new updates are made to a given single data item, eventually all accesses to that item will return the last updated value.

For most cases, eventual consistency is acceptable and provides better availability which is important. However, there are cases strong consistency is desired but ignored by developers that would lead to serious atomicity violation bugs.

4.3.4 Data Access via HTTP based API calls

In microservice, developers access data via HTTP interfaces and most projects follow RESTful architectural style when using HTTP interfaces. In the world of RESTful architectural style, a resource is a state representation of data and each resource is addressable at a unique path. A resource can have child resources, that is nested resources, for example, http://{host}/customers/{customerId}/balance has a resource that is a customer identified by customerId and a child resource named balance. For such RESTful design, we can detect all data access by checking URI (Uniform Resource Identifier) in the web traffic. And for those not exactly following RESTful design, such as the resource path in HTTP header or body rather than in HTTP URI, a specific rule can be defined accordingly to extract the resource path properly from web traffic. In this work, we focus on the data accesses that follows RESTful design.

4.3.5 Distributed Locks - Pessimistic Concurrency Control

One of the common problems found when building large scale distributed systems is how to ensure that only one process (or one server) across a cluster of servers access a resource. The resource can be a database, a file or data entries in a database. Without ACID guarantee by popular No-SQL databases, it is important to protect the accesses to data in the database that are shared by multiple processes or servers

when necessary. Actually, even with the ACID guarantee, there could be a need to protect the accesses across multiple transactions at the application level.

For many cases, eventual consistency is acceptable and would be the chosen model to provide better availability. However, there are cases where strong consistency is necessary. To achieve strong consistency of shared data, it needs a simple way to coordinate the execution of processes and ensure there is only one process accessing the resource at a time when needed. There are a lot of works on distributed lock management [47, 48, 49, 50], which is also known as pessimistic concurrency control [51]. When the atomicity of accesses to shared data is not enforced through distributed locks, it can be violated just like how atomicity violation manifests in multi-threaded programs.

As shown in Section 4.3.4, it is acceptable to assume the microservices are designed and implemented following RESTful way in which each resource has a unique URI to allow retrieval or updating. Similar to what is presented in Section 4.3.4 where each customer has a unique URI http://{host}/customers/{customerId} and the balance of a customer can be queried or updated at http://{host}/ customers/{customerId}/balance, the distributed lock also has its URI such as http://{host}/ locks/{lockId}, in which lockId can be its resource URI http://{host}/locks/customers/{customerId}. A HTTP POST operation is to acquire a lock and a HTTP Delete operation is to release the lock with the same lock Id.

4.3.6 Write-with-Version - Optimistic Concurrency Control

Distributed locks guarantee the strong consistency with the cost of performance because it takes extra time to acquire a lock and even more time to wait for a lock in case it has been acquired by others. It is applicable to all memory access pat-

terns. Write-with-version is the technique to guarantee strong consistency for certain memory access pattern that is read-then-write, without using distributed locks and its performance cost. Write-with-version is also known as optimistic concurrency control [52, 51].

To implement write-with-version, the data store is designed to provide a version number for each resource. Each time data is changed in resources, the version number changes. For instance, whenever a client retrieves data for a resource, it also receives the version of the resource since the version is part of the resource. And when a client performs an update, it provides the version of the resource that it is changing. If the provided version in the update request doesn't match the actual version of the resource in the data store, the update is rejected, typically with a HTTP error code 409 that means conflict, thus prevent inconsistency. It is ultimately the responsibility of client developers to deal with such update failure, which typically is reading again before write.

4.4 Predict Atomicity Violation in Distributed Systems

4.4.1 Overview of Our Method

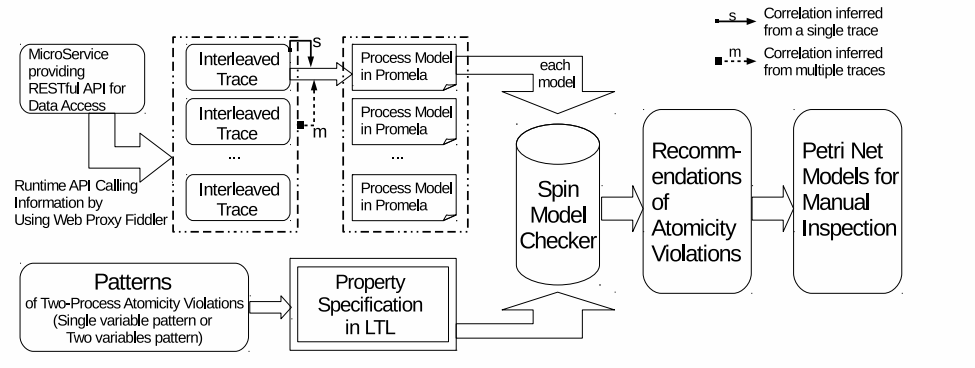
Based on the method using the existing McPatom with patterns of single variable atomicity violation, and another method in Chapter 3 extending McPatom with patterns of two variables atomicity violation, this section discusses a new method to predict atomicity violation in distributed systems, by applying what was learned from the above methods and studying the difference in distributed systems while there are similarities between protecting shared variables in multi-threaded programs

and shared data in distributed systems that run on multiple processes or multiple machines.

The framework contains the following major steps: (1) using Web Proxy Fiddler [53] to log web traffic of a microservice that provides RESTful API for data access and to record an interleaved trace containing only atomicity violation impacting events including all shared data accesses and all distributed lock accesses; (2) projecting the interleaved trace into a partial order process model of abstract processes, which maintains the causal relation within actual processes imposed by the distributed locks and write-with-version, and treats two correlated shared data as a single one using the patterns of single variable or keeps two correlated shared data using the patterns of two variables; (3) automatically translating the partial order process model into a Promela program for model checking in Spin [10]; (4) defining a complete set of atomicity violation patterns as in Figure 3.6 [7] involving a pair of processes accessing every single shared data and automatically translating them into temporal logic formulas; (5) defining a complete set of atomicity violation patterns involving a pair of processes accessing every pair of two shared data and automatically translating them into temporal logic formulas; (6) using Spin to model check the atomicity violation patterns; and (7) presenting the violation reported in Spin to a Petri net model for easy manual inspection.

Figure 4.1 gives an overview of the method. If using the patterns of two-thread atomicity violations with a single variable, the thread model in Promela treats the two correlated shared variables as a single one. If using the patterns of two-thread atomicity violations with two variables, the thread model in Promela keeps the two correlated shared variable.

Figure 4.1: Overview of the Method for Distributed Systems



4.4.2 Tracing the Execution of Microservices

Microservices run over HTTP in a sequence of RESTful API calls on a cluster of machines, we use a popular free web debugging proxy named Fiddler to capture all HTTP traffic from each machine in the cluster of the microservice, that includes all API calls to retrieve or update shared data, and acquire or release locks.

Each API call has a unique tracking Id to track all events during the API call from beginning to end. The tracking Id can be any Id that is unique in HTTP headers. A machine has concurrent API calls, and as a result, a cluster of machines also has concurrent API calls. Among those API calls, HttpGet is a reading of shared data, HttpPut is a write of shared data. For shared data, HttpPost and HttpDelete are out of consideration as they either create or delete resources and are not related to the typical cases of reading and writing the same resource. However, HttpPost is the one to acquire a lock due to its nature of creating a resource, and HttpDelete is the one to release a lock by its nature of deleting a resource.

Figure 4.2 shows a possible trace for the example in Program 4.1,

Figure 4.3 shows another trace of a microservice using distributed locks, in which "HttpPost http://localhost/locks/action"is a trace of acquiring a lock with lock Id

Figure 4.2: An Example of Trace

```
trackingId Event ResourceUri Response
trackingId-1 HttpGet http://localhost/accounts/123456/
   balance 200OK
trackingId-1 HttpPut http://localhost/accounts/123456/
   balance 200OK
trackingId-2 HttpGet http://localhost/accounts/123456/
   balance 200OK
trackingId-2 HttpPut http://localhost/accounts/123456/
   balance 200OK
```

"action" that is used to protect the resource with resource Id "action", and the corresponding "HttpDelete"is a trace of releasing the lock.

Figure 4.4 gives an example of the trace that involves write-with-version, which guarantee the strong consistency like distributed locks but without using distributed locks. The group of events by trackingId-1 simply read and write the resource successfully, however, in the group of events by trackingId-2 it get a response with HTTP status code 409Conflict and then read and write the same resource successfully, which is considered the pattern of write-with-version, essentially like using a distributed lock that guarantees the strong consistency between reads and writes. Figure 4.5 presents a visualization of the trace in an interleaving between two trackings that are on two processes or two machines.

4.4.3 Defining and Encoding Unserializable Interleaving Patterns between Two Processes

Based on the work of Chapter 3 that defines the unserializable interleaving patterns between two threads, this section proposes patterns to solve the problem between two processes in distributed systems with different challenges that are distributed

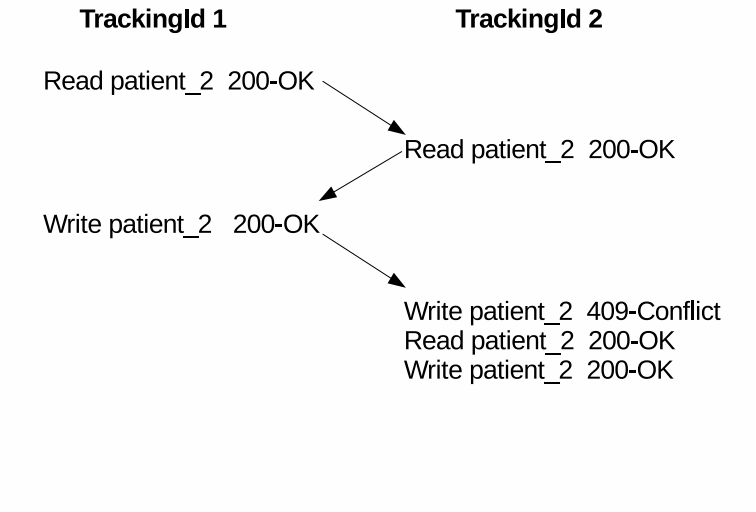
Figure 4.3: An Example of Trace With Locks

```
trackingId Event ResourceUri Response
trackingId-1 HttpPost http://localhost/locks/action 200OK
trackingId-1 HttpPut  http://localhost/action 200OK
trackingId-1 HttpDelete http://localhost/locks/action 200
   OK
trackingId-1 HttpPost http://localhost/locks/length 200OK
trackingId-1 HttpPut  http://localhost/length 200OK
trackingId-1 HttpDelete http://localhost/locks/length 200
   OK
trackingId-2 HttpPost http://localhost/locks/action 200OK
trackingId-2 HttpGet  http://localhost/action 200OK
trackingId-2 HttpDelete http://localhost/locks/action 200
   OK
trackingId-2 HttpPost http://localhost/locks/length 200OK
trackingId-2 HttpGet  http://localhost/length 200OK
trackingId-2 HttpDelete http://localhost/locks/length 200
   OK
```

Figure 4.4: An Example of Trace for Write-with-Version

```
trackingId Event ResourceUri Response
trackingId-1 HttpGet http://localhost:5984/main/
   patient_2_e9ecad62-b2f0-428a-8ecc-6797ef420d98 200OK
trackingId-2 HttpGet http://localhost:5984/main/
   patient_2_e9ecad62-b2f0-428a-8ecc-6797ef420d98 200OK
trackingId-1 HttpPut http://localhost:5984/main/
   patient_2_e9ecad62-b2f0-428a-8ecc-6797ef420d98 200OK
trackingId-2 HttpPut http://localhost:5984/main/
   patient_2_e9ecad62-b2f0-428a-8ecc-6797ef420d98 409
   Conflict
trackingId-2 HttpGet http://localhost:5984/main/
   patient_2_e9ecad62-b2f0-428a-8ecc-6797ef420d98 200OK
trackingId-2 HttpPut http://localhost:5984/main/
   patient_2_e9ecad62-b2f0-428a-8ecc-6797ef420d98 200OK
```

Figure 4.5: The Interleaving Pattern of Write-with-Version



locks and write-with-version, since write-with-version can be considered equivalent to a pair of read and write with proper distributed locks while preserving the semantics about strong consistency. To accommodate the cases of write-with-version, the pattern needs to take the API calling response code into consideration to identify the scenario of version conflict.

4.4.3.1 Unserializable Interleaving Patterns with Single Resource Involved

Figure 4.6 shows all possible scenarios of unserializable interleavings with only one access from Process 2. If any of the unserializable interleaving patterns is matched, it indicates a potential atomicity violation.

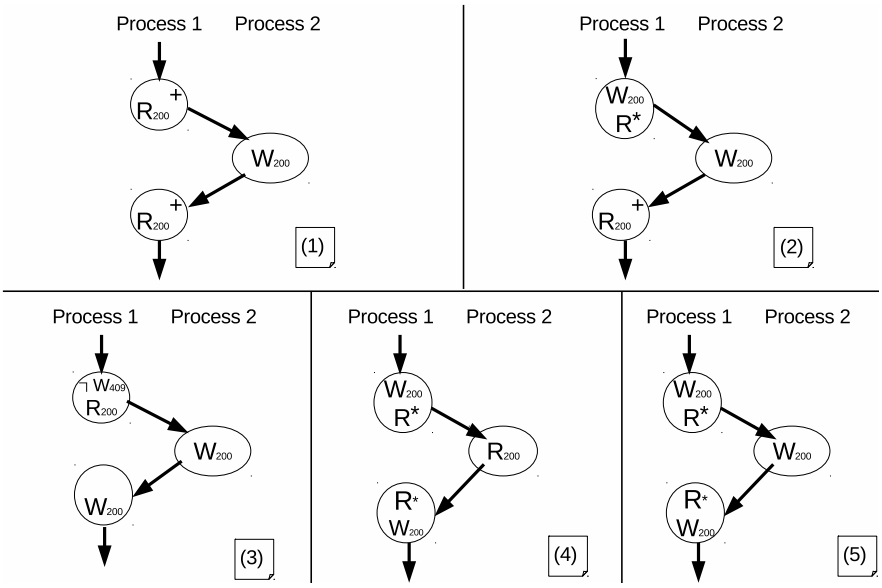


Figure 4.6: Unserializable Interleavings with two processes. In (1)(2)(3)(5), W in Process 2 unexpectedly changes the value; In (4), An intermediate value in Process 1 is read by Process 2. (3) is the pattern to recognize write-with-version as valid concurrency control, by making sure there is no writing returning conflict beforehand, marked as W_{409} since 409 is the HTTP status code for conflict. Other accesses marked R_{200} and W_{200} mean read with success and write with success correspondingly.

4.4.3.2 Unserializable Interleaving Patterns with Multiple Resources Involved

For the case with multiple resources involved, it is not affected by write-with-version which is for the same resource, so it is not necessary to check the response code of API calling, and Figure 3.9 can be simply reused.

4.4.4 Analyzing the Trace

The unique tracking Id of API calls can be used to group all events captured in Fiddler web proxy into API call processes in which each process has a unique tracking Id and contains multiple events about shared data and distributed locks.

4.4.4.1 Description of the Partial Order Process Model

For a multi-process microservice running R, an execution $\sigma = s_1, ..., s_n$ of is a sequence of executed accesses of resources. An execution can be projected to a sequence of annotated shared data accesses and synchronization events, which is the trace to analyze in this work. Formally, a trace, $\tau = e_1, ..., e_m$ is a sequence of events where each event $e_i (1 \leq i \leq m)$ is a tuple $\langle tid_i, timestamp_i, action_i, response_i \rangle$ in which tid_i is a tracking Id of API calls, $timestamp_i$ is a time stamp based on real time and $action_i$ is one of the following: (read/write, a shared resource) or (lock/unlock, a lock resource), and $response_i$ is the result of the action which is HTTP status code such as 200OK, 201Created, 409Conflict and etc. in the context of microservice based systems.

Given a trace $\tau = e_1, ..., e_m$ containing shared data accesses and lock events, a partial order process model $(E_\tau, \prec)$ is defined as follows:

1. $E_\tau = \{e_i \mid e_i \; in \; \tau\}$

   (a) $\prec$ is a partial order relation such that, for any $e_i, e_j \in E$ $(i \neq j)$, $e_i \prec e_j$ iff $tid_i = tid_j$ and $i < j$

   (b) Mutual exclusion: for any $e_i, e_j, e_m, e_n \in E$ $(i \neq j \neq m \neq n)$, $e_j \prec e_m$ or $e_n \prec e_i$ iff

      i. $tid_i = tid_j$, $action_i = (Lock, lvar)$, $action_j = (Unlock, lvar)$, and

      ii. $tid_m = tid_n$, $action_m = (Lock, lvar)$, $action_n = (Unlock, lvar)$ in which $lvar$ is a lock resource.

The partial order above defines the causal relation and is similar to the happened-before relation given in Section 3.3.1. The above definition ensures (1) shared data accesses within the same process are ordered, and (2) the constraint of lock is preserved regarding its impact to the shared data accesses across multiple processes. Therefore, it captures alternative traces that obey the same causal relation as τ and thus equivalent to the original trace, and each alternative trace τ' is a result of rearranging the order of some shared data accesses across different processes without breaking the constraint by $\prec$. The partial order process model enables exploration of all possible alternative traces that correspond to a set of feasible interleavings in a multi-process microservice running. However, the model provides an over-approximation without considering data-flow and dependencies between microservices processes, thus cannot guarantee each alternative trace captured in the model can be projected back to some feasible interleaved execution in the multi-process microservice running R, that is the reason for false positives.

4.4.4.2 Automatically Encoding Traces to Promela Code

The method in [7] to automatically encoding traces to Promela code can be modified by adding support of response code to build a partial order process model in Promela for each shared resource. And to check for atomicity violation involving multiple resources, the method in Section 3.4 can be applied to infer correlations between shared resources.

This work automatically encodes a trace to multiple Promela file with each one containing a single shared resource, or a pair of shared resources. It defines each shared resource v as a *short* in Promela, automatically assigns a unique value for all reading accesses and a unique value for all writing accesses in each thread. Formally, let $rw \in \{r, w\}$, and *tid* be tracking Id of the API calling, then v=*rw*+*tid* for each access of v. Our work sets r to be *0*, and w to be *10000* in case of failure such as 409Conflict and to be 20000 in case of success such as 200OK, 201Created. For example, given two tracking: *t1(tid=1)* and *t2(tid=2)*, and a shared resource v, our work makes assignments as below for each scenario of accesses.

- Assign *10000+1* to v for each writing access of v in tracking *t1* that has failing response 409Conflict,
- Assign *20000+1* to v for each writing access of v in tracking *t1* that has successful response 200OK,
- Assign *1* to v for each reading access of v in tracking *t1*,
- Assign *10000+2* to v for each writing access of v in tracking $t2$ that has failing response 409Conflict,
- Assign *20000+2* to v for each writing access of v in tracking $t2$ that has successful response 200OK,
- Assign *2* to v for each reading access of v in tracking *t2*.

```
proctype t1() { ... }
proctype t2()
{
  v_patient_2_e9ecad62-b2f0-428a-8ecc-6797ef420d98 = 2;
  v_patient_2_e9ecad62-b2f0-428a-8ecc-6797ef420d98 = 10002;
  v_patient_2_e9ecad62-b2f0-428a-8ecc-6797ef420d98 = 2;
  v_patient_2_e9ecad62-b2f0-428a-8ecc-6797ef420d98 = 20002;
}

init
{
  run t1();
  run t2();
  ...
}
```

Figure 4.7: A Sample of Partial Promela Code

Our work automatically generates Promela code for all synchronization primitives, like McPatom [7]. Figure 4.7 gives a sample of partial Promela code encoding a trace for Figure 4.4.

The resulting Promela code can be model checked with the patterns to predict alternative traces which is missing locks or write-with-version that are necessary to prevent atomicity violations. The following section discusses encoding the patterns into LTL formulas for the model checking tool Spin to use.

4.4.4.3 Automatically Encoding Atomicity Violation Patterns into Linear Time Temporal Logic (LTL) Formulas

For every shared resource and every pair of tracking $t1$ and $t2$, our work automatically defines a LTL formula (4.1) for each pattern in Figure 4.6 and another LTL formula (4.2) reversing the view of t_1 and t_2. Let v be a shared resource, $r = 0$, $w_{200} = 20000$ and $w_{409} = 10000$ as defined in section 4.4.4.2, $A_i \in \{r, w_{200}, w_{409}\}$,

and tid_i , $\overline{tid_i} \in \{1, 2\}$.

$$
\begin{aligned}
&[]! <> ((v == A_1 + tid_1)\&\& \\
&\quad X((v == A_2 + tid_2)U((v == A_3 + tid_3)\&\& \\
&\quad X((v == A_4 + tid_4)U(v == A_5 + tid_5))))) \\
&[]! <> ((v == A_1 + \overline{tid_1})\&\& \\
&\quad X((v == A_2 + \overline{tid_2})U((v == A_3 + \overline{tid_3})\&\& \\
&\quad X((v == A_4 + \overline{tid_4})U(v == A_5 + \overline{tid_5})))))
\end{aligned}
\tag{4.1}
\tag{4.2}
$$

where "[]" denotes *Always*, "!" denotes *Logical Negation*, "<>" denotes *Eventually*, "X" denotes *Next* and "U" denotes *Until*. These formulas specify that the atomicity violation patterns do not occur.

Using Figure 4.6 (3) as a concrete example, one formula in LTL is shown below:

$$
\begin{aligned}
&[]! <> ((!(v = w_{409} + 1)X(v == r + 1))\&\& \\
&\quad X((v == r + 1)U((v == w_{200} + 2)\&\& \\
&\quad X((v == w_{200} + 2)U(v == w_{200} + 1)))))
\end{aligned}
\tag{4.3}
$$

The predicted trace can be false positives and need to be examined to confirm. To make it easier to manually inspect the predictions, we propose a method to build a Petri net model from the predicted trace as in the following section.

4.4.4.4 Automatically Build a Petri Net Model From Predicted Trace

Petri net model is a natural choice to model the message passing, shared data and distributed locks that are all we need to analyze atomicity violation in distributed systems. Petri nets define elementary process steps as transitions that can model message passing and processing, and define data repositories as places. Each token in a place is a resource that could be shared data, and a transition consuming a token

from a place essentially locks the token and its representing shared data, a transition producing a token essentially unlocks the token and its representing shared data.

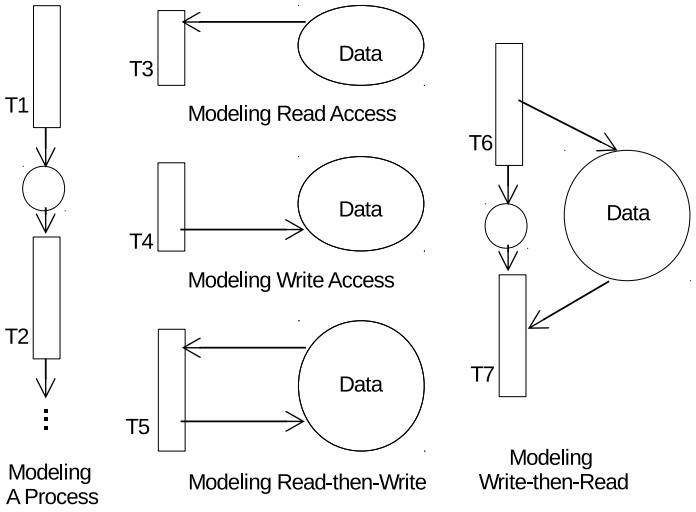
Modeling running of microservice is essentially modeling multiple processes in which each process has multiple sequential transitions. Each transition can access shared data including read access and write access, and for each shared data, its access in each process can be summarized as read, write, read-then-write, write-then-read. As shown in Figure 4.8, A process can have multiple transitions T1, T2 and more, their sequence is modeled by using a place between transitions. If it is a single read access T3 can be used and if it is a single write access, T4 can be used. In the case of read-then-write, a combination of T3 and T4 can be used like T5. When it is write-then-read, two transitions T6 and T7 are needed to model its sequence. A transition is atomic thus implying a proper distributed lock is used, so in case there is no distributed lock to guard multiple accesses, the corresponding transition can be labeled as "NonAtomic" to alert developers to check whether it is a case to add a distributed lock.

Our tool generates a simple text file describing the places, transitions and the edges between them, and use Graphviz [54] to visualize it. Petri nets in the following Figures are manually built to include annotations for easier understanding in this dissertation.

The automatically built Petri net model can be used to visually inspect a model to check whether the locking is sufficient to avoid atomicity violation.

For the example in Program 4.1, we build a Petri net model as shown in Figure 4.9 with manually annotations. Because of no distribution lock in place to guard its atomicity, we mark the transition in the Petri net model as "NonAtomic". In the model, $\varphi(AccountBalance) = \mathbb{P}(accountId \times balance)$ where an account is queried

Figure 4.8: Petri Net Modeling Method Overview



or updated by its accountId. Two transitions can be two "Deposit", two "Withdraw", or what is shown in 4.1 that is one "Deposit" and one "Withdraw".

There are cases that distributed lock is used but not enough or properly, as shown in the example of Figure 4.3. A Petri net model is built as in Figure 4.10, in which we don't mark any transition as "NonAtomic" as there are distributed locks being used to ensure those transitions being atomic. However, when we inspect the Petri

Figure 4.9: An Example of Petri Net Model

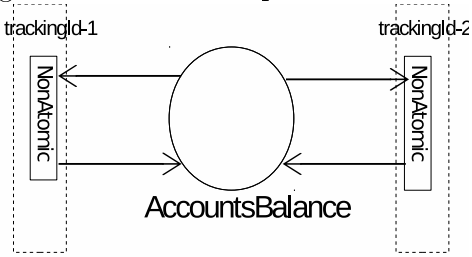
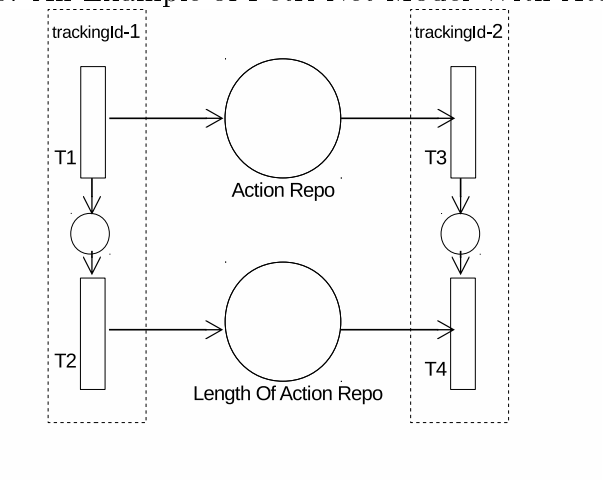


Figure 4.10: An Example of Petri Net Model With Atomic Transitions



net model, it is clear that there is a possible atomicity violation if the transitions are fired in the order: T1, T3, T4, T2 which essentially read inconsistent intermediate states.

4.5 Evaluation

All of our experiments are conducted on a machine with 2 Core 2.3GHz CPU, 4GB of memory, running Debian 8.7 as the operating system. We use Fiddler v5.0 for web traffic logging and SPIN tool 6.4.6 for model checking. Our experiments discover two bugs that were not known.

4.5.1 HospitalRun: an open source electronic medical record system

HospitalRun [55] is a freely available modern software platform for developing world hospitals, that uses CouchDB [56] as the underlying NoSQL data store through its

HTTP based RESTful API. It has been forked more than 1000 times in Github by other developers and has 233 thousands lines of code. The project is based on Node.js and we are able to configure Node.js to use the proxy set up by Fiddler Web Proxy so that it captures all HTTP/HTTPS traffic to and from the underlying CouchDB.

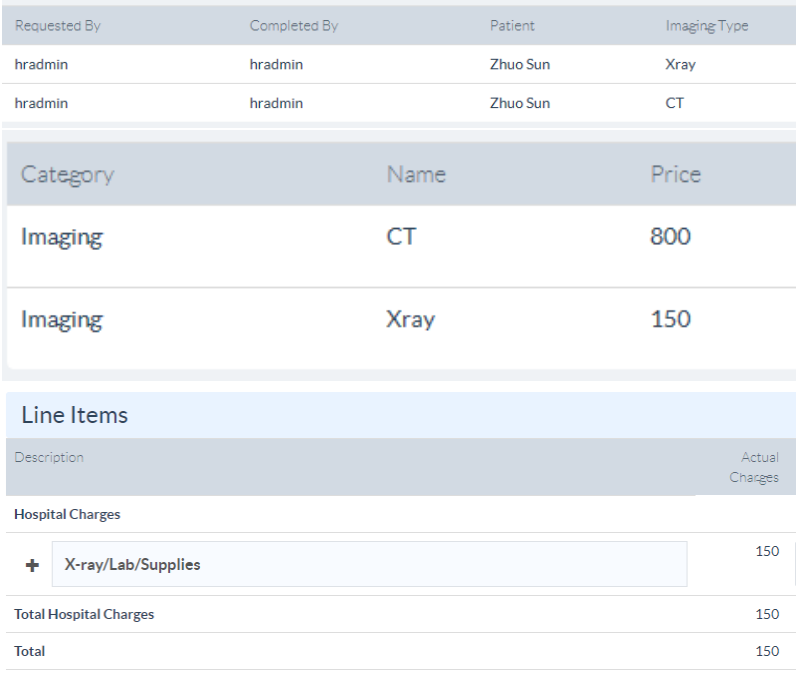
Our tool captured a trace of 292KB and found an atomicity violation that was not reported and we can reproduce. It took about 2 seconds to infer the correlation between shared resources and less than one minute for Spin model checkers to check all properties for a shared resource or a pair of shared resources. Following are the relevant trace when trying to reproduce the predicted atomicity violation. HttpPost was used to query resources with multiple resource keys to query in the HTTP body. We had to define a special rule to interpret a HttpPost on the API endpoint /main/_all_docs?include_docs=true to be a reading event with the resource to be read in the HTTP body. As shown in Figure 4.11, it read the resource visit_2_ff493467-7087-407d-b9f9-f0cc03fbc167 then wrote the resource billingLineItem _2_13f5c098-b31c-4e70-ae4a-b5f92792c725 in tracking 1, which is meant to query the list of imaging requested for a visit then generate a bill item for the visit based on all imaging requests. It got one imaging request and then generated a bill item for the invoice. However, before generating the invoice, another API came in to update the resource visit_2_ff493467-7087-407d-b9f9-f0cc03fbc167 with an extra imaging request that resulted in two imaging requests for the visit to charge, so that the invoice missed the new imaging request and became a loss for the hospital.

It can happen when the administrator open the invoice page and left for a while so that upon coming back to the invoice page and generate an invoice it triggered the atomicity violation bug. As shown in Figure 4.12, it has two imaging request

Figure 4.11: A Sample of Partial Trace

```
POST http://localhost:5984/main/_all_docs?include_docs=
   true HTTP/1.1
{"keys":["visit_2_ff493467-7087-407d-b9f9-f0cc03fbc167"]}
user-agent: 1
HTTP/1.1 200 OK
{"id":"visit_2_ff493467-7087-407d-b9f9-f0cc03fbc167","key
   ":"visit_2_ff493467-7087-407d-b9f9-f0cc03fbc167","
   value":{"imaging":["0df33a57-2149-43ef-a72d-2947
   b1e3e6d7"],...}}}}

PUT http://localhost:5984/main/visit_2_ff493467-7087-407d
   -b9f9-f0cc03fbc167 HTTP/1.1
{"_rev":"2-36bf667062d2faf0f1a963b0e4747b1f","_id":"
   visit_2_ff493467-7087-407d-b9f9-f0cc03fbc167","data
   ":{"imaging":["0df33a57-2149-43ef-a72d-2947b1e3e6d7
   ","5636ab7c-99a1-4e26-99df-55cc05cb4556"]...}}}
user-agent: 2
HTTP/1.1 201 Created

PUT http://localhost:5984/main/billingLineItem_2_13f5c098
   -b31c-4e70-ae4a-b5f92792c725 HTTP/1.1
{"_rev":"1-6ebb79c0cf3869dd521d561fca2000da","_id":"
   billingLineItem_2_13f5c098-b31c-4e70-ae4a-b5f92792c725
   ","data":{"amountOwed":150,...}}}
user-agent: 1
HTTP/1.1 201 Created
```

Figure 4.12: Screenshot when reproducing the predicted atomicity violation



Requested By	Completed By	Patient	Imaging Type
hradmin	hradmin	Zhuo Sun	Xray
hradmin	hradmin	Zhuo Sun	CT

Category	Name	Price
Imaging	CT	800
Imaging	Xray	150

Line Items

Description	Actual Charges
Hospital Charges	
+ X-ray/Lab/Supplies	150
Total Hospital Charges	150
Total	150

including one Xray and one CT, according to the pricing, the total should be $950, however, the generated invoice is $150 because it is missing one imaging request caused by the atomicity violation.

Our tool found the two shared resources visit_2_ff493467-7087-407d-b9f9-f0cc03fbc167 and billingLineItem_2_13f5c098-b31c-4e70-ae4a-b5f92792c725 are correlated because they often show up together in the trace which is as expected since the bill item is for the correlated visit. Then for each pair of correlated resources, our tool generated Promela code that includes one example as shown in Figure 4.13.

The model checking tool Spin gave a trace of predicted atomicity violation, according to the pattern (1) in Figure 3.9, based on that, our tool generated a Petri net model as shown in Figure 4.14, in which the type is defined as below with Id being a string for the unique Id and $Value$ being a string for the value of the resource specified by the Id. Each place is a power set that allows to query the existence of a Id and update the value for a Id. When the transition is fired in the order of T1,

Figure 4.13: Partial Promela Code of the HospitalRun trace

```
proctype t1()
{
...
  v_visit_2_ff493467-7087-407d-b9f9-f0cc03fbc167 = 1;
  v_billingLineItem_2_13f5c098-b31c-4e70-ae4a-
     b5f92792c725 = 20001;
...
}
proctype t2()
{
...
  v_visit_2_ff493467-7087-407d-b9f9-f0cc03fbc167 = 20002;
...
}

init
{
  run t1();
  run t2();
}
```

T2 and T3, it simulates the predicted atomicity violation.

$$\varphi(Visit) = \mathbb{P}(Id \times Value)$$

$$\varphi(BillingLineItem) = \mathbb{P}(Id \times Value)$$

$$R(T1) = \exists\,(v \in Visit)\,.\,(\; \frac{v[1] = VisitIdToQuery}{\wedge\, Visit' = Visit}\;)$$

$$R(T2) = \exists\,(v \in Visit)\,.\,(\; \frac{v[1] = VisitIdToQuery}{\wedge\, Visit' = Visit \setminus \{v\} \cup \{(v[1], visitIdNewValue)\}}\;)$$

$$R(T3) = \exists\,(b \in BillingLineItem)\,.\,(\; \frac{b[1] = BillingLineItemIdToQuery}{\wedge\, BillingLineItem' = BillingLineItem \setminus \{b\} \cup \{(b[1], NewValue)\}}\;)$$

4.5.2 Google Cloud Storage FUSE: A user-space file system for interacting with Google Cloud Storage

Google Cloud Storage FUSE (GCS-FUSE) is a Google-developed open source FUSE adapter that allows to mount Google Cloud Storage buckets as file systems on Linux

Figure 4.14: Petri net model for the predicted atomicity violation

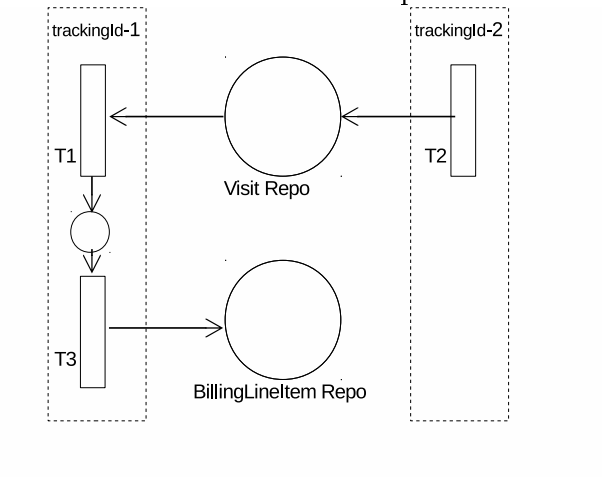


Figure 4.15: Examples of Google Cloud Storage JSON API

```
get         GET /b/bucket/o/object       Retrieves an
   object or its metadata.
insert  POST /upload/storage/v1/b/bucket/o      Stores a
   new object and metadata.
insert    POST /b/bucket1/o/object1/copyTo/bucket2/o/
   object2 Copies an existing object to another object
update  PUT  /b/bucket/o/object Updates an object's
   metadata.
delete  DELETE  /b/bucket/o/object      Deletes an object
    and its metadata.
```

or macOS systems. It is based on Google Cloud Storage JSON API which is RESTful as shown in a few example API calls in Figure 4.15.

Here are a few steps to facilitate capturing web traffic of GCS-FUSE.

1. Copy root certificate generated by web proxy to /etc/ssl/certs/
2. export http_proxy=127.0.0.1:8888
3. export https_proxy=127.0.0.1:8888
4. Run GCS-FUSE in foreground mode:

   gcsfuse --foreground gcs-fuse-av ~/projects/gcs-fuse/

Our tool captured a trace of 224KB and found an atomicity violation that was not reported and we can reproduce. It took less than one minute for Spin model checkers to check all properties. Following Figure 4.16 shows the relevant traces. GCS-FUSE uses a generation number to represent the version of a file inode in the file system, and increase the generation whenever making a change to the file inode. When uploading a file to the existing file inode, GCS-FUSE uploads the file to a temporary stream with a bigger generation number and copies the temporary stream to the destination file inode as shown in the API "/copyTo/b/{blockName}/o/{objectName}" of Figure 4.16.

GCS-FUSE doesn't follow exactly RESTful API design about the resource URI, but we can define simple rules to figure out object names from the URIs according to the pattern shown in Figure 4.15. For the object as shared resource, our tool generated Promela code that includes one example as shown in Figure 4.17.

The model checking tool Spin gave a trace of predicted atomicity violation, according to the pattern (3) in Figure 4.6, based on that, our tool generated a Petri net model as shown in Figure 4.18, in which the type is defined as below with Id

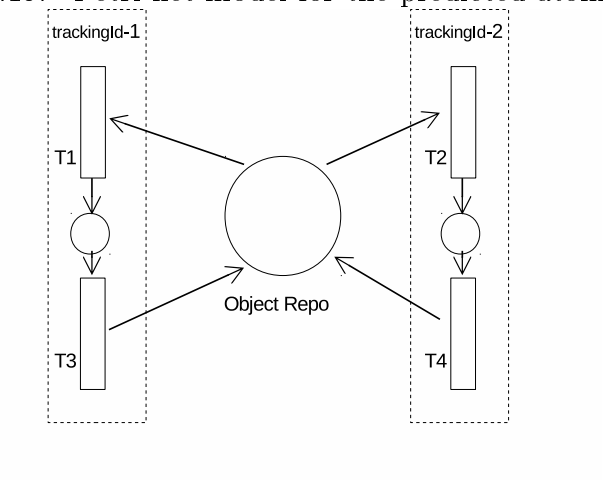
Figure 4.16: A Sample of Partial Trace of GCS-FUSE

```
GET https://www.googleapis.com/download/storage/v1/b/gcs-
   fuse-av/o/atom.bib?alt=media&generation
   =1540133382180427 HTTP/1.1
user-agent: 1
HTTP/1.1 200 OK

POST https://www.googleapis.com/storage/v1/b/gcs-fuse-av/
   o/.goutputstream-BMRWQZ/copyTo/b/gcs-fuse-av/o/atom.
   bib?ifSourceMetagenerationMatch=1&projection=full&
   sourceGeneration=1540135196194955 HTTP/1.1
user-agent: 1
HTTP/1.1 200 OK

GET https://www.googleapis.com/download/storage/v1/b/gcs-
   fuse-av/o/atom.bib?alt=media&generation
   =1540133382180427 HTTP/1.1
user-agent: 2
HTTP/1.1 200 OK

POST https://www.googleapis.com/storage/v1/b/gcs-fuse-av/
   o/.goutputstream-3RY5QZ/copyTo/b/gcs-fuse-av/o/atom.
   bib?ifSourceMetagenerationMatch=1&projection=full&
   sourceGeneration=1540135201098064 HTTP/1.1
user-agent: 2
HTTP/1.1 200 OK
```

Figure 4.17: Partial Promela Code of the GCS-FUSE trace

```
proctype t1()
{
...
  v_atom_bib = 1;
  v_atom_bib = 20001;
...
}
proctype t2()
{
...
  v_atom_bib = 2;
  v_atom_bib = 20002;
...
}

init
{
  run t1();
  run t2();
}
```

Figure 4.18: Petri net model for the predicted atomicity violation



being a string for the unique Id and $Value$ being a string for the value of the resource specified by the Id. The place is a power set that allows to query the existence of a Id and update the value for a Id. When the transition is fired in the order of T1, T2, T3, and T4, it simulates the predicted atomicity violation.

$$\varphi(Object) = \mathbb{P}(Id \times Value)$$

4.6 Related Works

DCatch [57] predicted distributed concurrency bugs including atomicity violation bugs by analyzing the correct execution of distributed systems. It designed a set of happens-before rules, runtime tracing tools and trace analysis tools. Its focus was on the infrastructure software that helps to run a distributed system. Our work focuses on the applications that are run on top of the infrastructure software, that are designed in microservice based architecture style and uses RESTful API for shared data access.

In [58] a formal framework is presented based on Petri nets for the process flow in microservices architecture style. It helped to analyze the process flow defined in an orchestration engine that is in charge of enacting a script to define the high-level control and data flows. It can be applied to the specific orchestration languages to model the communication between microservices that are defined by the orchestration. However, in the real world microservices are much more than those ones defined in orchestration. Our work is flexible to model the communication between any microservices.

In [59] it was investigated how to extract a process model from system event logs. The research area of process mining focused on extracting information about processes by checking system event logs including which activities are performed, at what time, by whom and in the context of which case (i.e., process instance). By explicitly using the case context, process discovery algorithms are capable of constructing process models that accurately describe the process [59]. Our captured web traffic about API calls in microservices is similar to the interested data in system event logs, including which API is called, at what time, on what resources and in the context of which tracking Id. Our extracted Petri nets models accurately describe the shared data, their accesses, and the related distributed locks.

4.7 Summary

Distributed concurrency bugs often have really simple causes and can be caught by simple tests [44], however, they are extremely hard to troubleshoot and detect due to its complex non-deterministic nature. As the most common distributed concurrency bugs, atomicity violations are studied in recent works [44, 57, 4]. This paper presents

a tool based on McPatom [7] using model checking to predict atomicity violation distributed concurrency bugs, in the microservice based modern distributed systems.

The tool is powerful and it is able to capture runtime trace of shared resources and infer the correlation between shared resources. The tool can explore a vast interleaving space of a microservice based modern distributed system given a small set of captured test runs. It is applicable to large real-world systems and predicts an atomicity violation in a popular open source project.

CHAPTER 5

CONCLUSION

5.1 Summary

This dissertation presents methods and tools for modeling and analyzing concurrent software systems that run as multi-threaded programs or microservice based distributed systems, to predict atomicity violation bugs and improve the reliability of concurrent software. For multi-threaded programs, we use binary instrumentation tool to capture runtime information about shared variables and synchronization events, and for microservice based distributed systems, we use a web proxy to capture HTTP based traffic about API calls and the shared resources they access including distributed locks. Based on the capture traces, we develop methods to extract a partial order model and apply model checking techniques by defining the atomicity violation patterns in LTL formulas. We also develop methods to infer the correlation between shared variables in multi-threaded programs and shared resources in microservice based distributed systems and define patterns for multi-variable atomicity violation.

Our tool using model checking to predict atomicity violation concurrency bugs is powerful and can explore a vast interleaving space of non-deterministic programs including multi-threaded programs and microservice based distributed systems, given a small set of captured test runs. Our tool is applicable to large real-world systems. The experiment result shows the scalability of our methods is promising compared to related works and our methods are able to detect well-known atomicity violations in multi-threaded programs and also discover a couple of new atomicity violations in real-world microservice based distributed systems.

5.2 Future Work

For the microservice based distributed systems, we predict atomicity violations and present them in Petri net models. The Petri net models are small ones only covering the relevant resources in a relevant pattern. Additional methods can be developed based on existing process mining algorithms to give a larger Petri net model covering more resources and better picture of the overall systems, that makes it possible to do more analysis based on Petri net models. And the partial order process model we define for microservice based distributed systems doesn't take barriers into consideration as they are rarely used in microservice based distributed systems, can be enhanced to add barriers to reduce false positives when barriers are used in the microservice based distributed system.

We can also support the logging framework chosen by the authors of the microservice such as log4j and log4j2, to adapt to more distributed systems that might have more complicated data design and not follow RESTful API and resource design. Using logging frameworks also make it more flexible to get more information between correlated variables and resources so that it can improve the confidence when inferring the correlation between them thus helping to find the real atomicity violations.

BIBLIOGRAPHY

[1] K. Poulsen, "Software bug contributed to blackout." URL: `http://www.securityfocus.com/news/8016`, 2004. [Online; Accessed: 07/16/2011].

[2] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "AVIO: detecting atomicity violations via access interleaving invariants," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, (San Jose, CA, USA), pp. 37–48, 2006.

[3] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou, "MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs," *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (SOSP '07)*, pp. 103–116, Oct. 2007.

[4] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi, "Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, (New York, NY, USA), pp. 517–530, ACM, 2016.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*, vol. 5. Addison-wesley New York, 1987.

[6] S. Lu, S. Park, and Y. Zhou, "Finding Atomicity-Violation bugs through unserializable interleaving testing," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, p. 1, 2011.

[7] R. Zeng, Z. Sun, S. Liu, and X. He, "McPatom: a predictive analysis tool for atomicity violation using model checking," in *Proceedings of the 19th international conference on Model Checking Software (SPIN'12)*, (Oxford, UK), pp. 191–207, 2012.

[8] C. Wang, R. Limaye, M. Ganai, and A. Gupta, "Trace-based symbolic analysis for atomicity violations," in *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'10)*, (Paphos, Cyprus), pp. 328–342, 2010.

[9] A. Pnueli, "The Temporal Logic of Programs," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, (Washington, DC, USA), pp. 46–57, IEEE Computer Society, 1977.

[10] G. Holzmann, *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[11] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, "Simple On-the-fly Automatic Verification of Linear Temporal Logic," in *Protocol Specification, Testing and Verification XV: Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland, June 1995* (P. Dembiński and M. Średniawa, eds.), IFIP Advances in Information and Communication Technology, pp. 3–18, Boston, MA: Springer US, 1996.

[12] J. Esparza and K. Heljanko, *Unfoldings: a partial-order approach to model checking*. Monographs in theoretical computer science: an EATCS series, Berlin: Springer, 2008. OCLC: ocn209334163.

[13] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[14] A. Valmari, "The state explosion problem," in *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets* (W. Reisig and G. Rozenberg, eds.), Lecture Notes in Computer Science, pp. 429–528, Berlin, Heidelberg: Springer Berlin Heidelberg, 1998.

[15] A. Valmari, "A Stubborn Attack on State Explosion," *Form. Methods Syst. Des.*, vol. 1, pp. 297–322, Dec. 1992.

[16] D. Peled, "All from one, one for all: on model checking using representatives," in *Computer Aided Verification* (C. Courcoubetis, ed.), Lecture Notes in Computer Science, pp. 409–423, Springer Berlin Heidelberg, 1993.

[17] K. L. McMillan, "Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits," in *Proceedings of the Fourth International Workshop on Computer Aided Verification*, CAV '92, (London, UK, UK), pp. 164–177, Springer-Verlag, 1993.

[18] D. Peled, "Combining partial order reductions with on-the-fly model-checking," *Formal Methods in System Design*, vol. 8, pp. 39–64, Jan. 1996.

[19] R. P. Kurshan, "Reducibility in analysis of coordination," in *Discrete Event Systems: Models and Applications* (P. Varaiya and A. B. Kurzhanski, eds.), Lecture Notes in Control and Information Sciences, pp. 19–39, Springer Berlin Heidelberg, 1988.

[20] J.-C. Fernandez, L. Mounier, C. Jard, and T. Jéron, "On-the-fly verification of finite transition systems," *Formal Methods in System Design*, vol. 1, pp. 251–273, Oct. 1992.

[21] G. J. Holzmann, *Design and Validation of Computer Protocols*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1991.

[22] X. He, "Modeling and Analyzing Cyber Physical Systems Using High Level Petri Nets," in *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 469–476, July 2018.

[23] H. J. Genrich, "Predicate/Transition Nets," in *Petri Nets: Central Models and Their Properties* (W. Brauer, W. Reisig, and G. Rozenberg, eds.), Lecture Notes in Computer Science, pp. 207–247, Springer Berlin Heidelberg, 1987.

[24] Z. Sun, R. Zeng, and X. He, "A Method for Predicting Two-Variable Atomicity Violations," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 103–110, July 2018.

[25] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, pp. 558–565, July 1978.

[26] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *the 2005 ACM Conference on Programming Language Design and Implementation (PLDI'05)*, (Chicago, IL, USA), pp. 190–200, 2005.

[27] C. Leys, C. Ley, O. Klein, P. Bernard, and L. Licata, "Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median," *Journal of Experimental Social Psychology*, vol. 49, pp. 764–766, July 2013.

[28] C. Flanagan, S. N. Freund, and J. Yi, "Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs," in *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, (Tucson, AZ, USA), pp. 293–303, 2008.

[29] B. Lucia, L. Ceze, and K. Strauss, "ColorSafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations," in *Proceedings of the 37th annual international symposium on Computer architecture - ISCA '10*, (Saint-Malo, France), p. 222, 2010.

[30] S. Park, R. Vuduc, and M. J. Harrold, "Unicorn: a unified approach for localizing non-deadlock concurrency bugs," *Software Testing, Verification and Reliability*, vol. 25, no. 3, pp. 167–190, 2015.

[31] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*, (San Diego, CA, USA), pp. 267–280, 2008.

[32] L. Wang and S. D. Stoller, "Runtime analysis of atomicity for multithreaded programs," *IEEE Transactions on Software Engineering*, vol. 32, pp. 93–110, 2006.

[33] F. Sorrentino, A. Farzan, and P. Madhusudan, "Penelope: weaving threads to expose atomicity violations," in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)*, (Santa Fe, NM, USA), pp. 37–46, 2010.

[34] A. Farzan and P. Madhusudan, "The complexity of predicting atomicity violations," in *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, (York, UK), pp. 155–169, 2009.

[35] F. Chen, T. F. Serbanuta, and G. Rosu, "jPredictor: a predictive runtime analysis tool for java," in *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, (Leipzig, Germany), pp. 221–230, 2008.

[36] T. Serbanuta, F. Chen, and G. Rosu, "Maximal causal models for sequentially consistent systems," in *Proceedings of the 3rd International Conference on Runtime Verification (RV'12)*, (Istanbul, Turkey), pp. 136–150, 2012.

[37] K. Sen, G. Rosu, and G. Agha, "Detecting errors in multithreaded programs by generalized predictive analysis of executions," in *Proceedings of the 7th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)*, (Athens, Greece), pp. 211–226, 2005.

[38] A. Sinha, S. Malik, C. Wang, and A. Gupta, "Predictive analysis for detecting serializability violations through trace segmentation," in *Proceedings of the 9th International Conference on Formal Methods and Models for Codesign (MEMOCODE'11)*, (Cambridge, UK), pp. 99–108, 2011.

[39] V. Kahlon and C. Wang, "Universal causality graphs: a precise happens-before model for detecting bugs in concurrent programs," in *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'10)*, (Edinburgh, United Kingdom), pp. 434–449, 2010.

[40] M. K. Ganai, "Scalable and precise symbolic analysis for atomicity violations," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*, (Lawrence, KS, USA), pp. 123–132, 2011.

[41] J. Yi, C. Sadowski, and C. Flanagan, "SideTrack: generalizing dynamic atomicity analysis," in *Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD'09)*, (Chicago, IL, USA), pp. 8:1–8:10, 2009.

[42] A. Jannesari and F. Wolf, "Automatic generation of unit tests for correlated variables in parallel programs," *International Journal of Parallel Programming*, vol. 44, pp. 644–662, June 2016.

[43] W. Ahmed and Y. W. Wu, "A survey on reliability in distributed systems," *Journal of Computer and System Sciences*, vol. 79, pp. 1243–1255, Dec. 2013.

[44] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. Jain, and M. Stumm, "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems.," in *OSDI*, pp. 249–265, 2014.

[45] E. A. Brewer, "Towards Robust Distributed Systems (Abstract)," in *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, (New York, NY, USA), pp. 7–, ACM, 2000.

[46] S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services," *SIGACT News*, vol. 33, pp. 51–59, June 2002.

[47] P. Schwan *et al.*, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the 2003 Linux symposium*, vol. 2003, pp. 380–386, 2003.

[48] W. E. Snaman and D. W. Thiel, "The vax/vms distributed lock manager," *Digital Technical Journal*, vol. 5, no. 2, p. 44, 1987.

[49] K. Thomas, "Programming locking applications," *IBM Corporation*, vol. 2, no. 00, p. 1, 2001.

[50] F. Junqueira and B. Reed, *ZooKeeper: distributed process coordination.* " O'Reilly Media, Inc.", 2013.

[51] D. A. Menascé and T. Nakanishi, "Optimistic versus pessimistic concurrency control mechanisms in database management systems," *Information Systems*, vol. 7, pp. 13–27, Jan. 1982.

[52] H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Trans. Database Syst.*, vol. 6, pp. 213–226, June 1981.

[53] "Fiddler - free web debugging proxy." URL: `https://www.telerik.com/fiddler`. [Online; Accessed: 08/26/2018].

[54] "Graphviz - graph visualization software." URL: `https://www.graphviz.org/`. [Online; Accessed: 08/26/2018].

[55] "Hospitalrun - freely available, modern software platform for developing world hospitals." URL: `https://cure.org/2016/01/cure-international-offers-freely-available-open-source-software-platform-to-developing-world-hospitals/`. [Online; Accessed: 08/26/2018].

[56] "Apache couchdb - nosql data store with an intuitive http/json api and designed for reliability." URL: `http://couchdb.apache.org`. [Online; Accessed: 08/26/2018].

[57] H. Liu, G. Li, J. F. Lukman, J. Li, S. Lu, H. S. Gunawi, and C. Tian, "Dcatch: Automatically detecting distributed concurrency bugs in cloud systems," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, (New York, NY, USA), pp. 677–691, ACM, 2017.

[58] M. Camilli, C. Bellettini, L. Capra, and M. Monga, "A Formal Framework for Specifying and Verifying Microservices Based Process Flows," in *Software Engineering and Formal Methods* (A. Cerone and M. Roveri, eds.), Lecture Notes in Computer Science, pp. 187–202, Springer International Publishing, 2018.

[59] B. F. Dongen, A. K. A. D. Medeiros, and L. Wen, "Process mining: Overview and outlook of petri net discovery algorithms," in *Transactions on Petri Nets*

and Other Models of Concurrency II: Special Issue on Concurrency in Process-Aware Information Systems, pp. 225–242, Springer-Verlag, 2009.

PUBLICATIONS BY ZHUO SUN

[1] Zhuo Sun, Reng Zeng, and Xudong He. A Method for Predicting Two-Variable Atomicity Violations. In *18th IEEE International Conference on Software Quality, Reliability and Security, QRS 2018, Lisbon, Portugal, July 16-20, 2018*, pages 103–110, 2018.

[2] Xudong He, Reng Zeng, Su Liu, Zhuo Sun, and Kyungmin Bae. A term rewriting approach to analyze high level petri nets. In *10th International Symposium on Theoretical Aspects of Software Engineering, TASE 2016, Shanghai, China, July 17-19, 2016*, pages 109–112, 2016.

[3] Reng Zeng, Zhuo Sun, Su Liu, and Xudong He. A method for improving the precision and coverage of atomicity violation predictions. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 116–130, 2015.

[4] Su Liu, Reng Zeng, Zhuo Sun, and Xudong He. Bounded model checking high level petri nets in pipe+verifier. In *Formal Methods and Software Engineering - 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings*, pages 348–363, 2014.

[5] Reng Zeng, Zhuo Sun, Su Liu, and Xudong He. Mcpatom: A predictive analysis tool for atomicity violation using model checking. In *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings*, pages 191–207, 2012.

[6] Su Liu, Reng Zeng, Zhuo Sun, and Xudong He. SAMAT - A tool for software architecture modeling and analysis. In *Proceedings of the 24th International Conference on Software Engineering & Knowledge Engineering (SEKE'2012), Hotel Sofitel, Redwood City, San Francisco Bay, USA July 1-3, 2012*, pages 352–358, 2012.

[7] Zhuo Sun, Masoumeh Karimi, Deng Pan, Zhenyu Yang, and Niki Pissinou. Buffered crossbar based parallel packet switch. In *Proceedings of the Global Communications Conference, 2010. GLOBECOM 2010, 6-10 December 2010, Miami, Florida, USA*, pages 1–5, 2010.

[8] Masoumeh Karimi, Zhuo Sun, Deng Pan, and Zhenyu Yang. Reducing crosspoint buffers for performance guaranteed asynchronous crossbar scheduling. In *Proceedings of the Global Communications Conference, 2010. GLOBECOM 2010, 6-10 December 2010, Miami, Florida, USA*, pages 1–5, 2010.

[9] Yechang Fang, Kang Yen, Deng Pan, and Zhuo Sun. Buffer management algorithm design and implementation based on network processors. *CoRR*, abs/1005.0905, 2010.

[10] Masoumeh Karimi, Zhuo Sun, Deng Pan, and Zesheng Chen. Packet-mode asynchronous scheduling algorithm for partially buffered crossbar switches. In *Proceedings of the Global Communications Conference, 2009. GLOBECOM 2009, Honolulu, Hawaii, USA, 30 November - 4 December 2009*, pages 1–6, 2009.

VITA

ZHUO SUN

1998–2002	B.E., Computer Science Guangxi University Nanning, China
2003–2005	M.E., Software Engineering Sun Yat-Sen University Guangzhou, China
2005–2007	Software Quality Engineer Nortel Networks Guangzhou, China
2008–2011	Master, Telecommunication Florida International University Miami, Florida
2011–2018	Doctoral Candidate, Computer Science Florida International University Miami, Florida
2016–2018	Software Automation Engineer Motorola Solutions Plantation, Florida