Software Qual J (2018) 26:855–889
DOI 10.1007/s11219-017-9385-3

A systematic survey on automated concurrency bug detection, exposing, avoidance, and fixing techniques

Haojie Fu[1] · Zan Wang[1] · Xiang Chen[2] · Xiangyu Fan[1]

Published online: 10 August 2017

Abstract Currently, concurrent programs are becoming increasingly widespread to meet the demands of the rapid development of multi-core hardware. However, it could be quite expensive and challenging to guarantee the correctness and efficiency of concurrent programs. In this paper, we provide a systematic review of the existing research on fighting against concurrency bugs, including automated concurrency bug exposing, detection, avoidance, and fixing. These four categories cover the different aspects of concurrency bug problems and are complementary to each other. For each category, we survey the motivation, key issues, solutions, and the current state of the art. In addition, we summarize the classical benchmarks widely used in previous empirical studies and the contribution of active research groups. Finally, some future research directions on concurrency bugs are recommended. We believe this survey would be useful for concurrency programmers and researchers.

Keywords Concurrency bug detection · Concurrency bug exposing · Concurrency bug avoidance · Concurrency bug fixing

✉ Zan Wang
wangzan@tju.edu.cn

Haojie Fu
fhj0909@tju.edu.cn

Xiang Chen
xchencs@ntu.edu.cn

Xiangyu Fan
fxy@tju.edu.cn

1 School of Computer Software, Tianjin University, Tianjin, People's Republic of China

2 School of Computer Science and Technology, Nantong University, Nantong, People's Republic of China

1 Introduction

Over the past few decades, the increasing availability of multi-core hardware is making concurrent programs ubiquitous in the industrial and academic research area. However, it also raises new challenges such as the well-known concurrency bug, which is introduced by improper coordination among multi-threads and has caused severe failures with huge financial loses and even loss of life. To reduce these losses and improve the reliability of concurrent programs, an increasing number of researchers have proposed numerous testing approaches to detect, expose, and fix concurrent bugs.

As the preliminary step of bug detection, some research focuses on how to expose concurrency bugs effectively and thoroughly. For some bugs, which are unable to completely or unnecessarily be fixed, researchers have explored various ways to avoid them or to recover the programs from those bugs without actually fixing them. However, concurrent software quality has not been improved before concurrency bugs are fixed. Therefore, another research topic of whether concurrency bugs can be fixed automatically has been raised recently. Cost-effective automatic repair techniques could improve the concurrent programs' quality at relatively low cost while significantly reducing losses brought by defects. Nevertheless, a number of challenges and open issues towards this goal remain unsolved.

1.1 Review method

This survey provides a systematic review of existing research on concurrency bugs, including their exposing, detection, avoiding, and fixing. We reviewed the existing literature by searching through journal articles and conference papers addressing any topic related to concurrency bugs, including methods, tools, techniques, empirical evaluations, and surveys. The search for relevant papers was carried out in the online repositories. Firstly, we performed the search in the Google Scholar, DBLP database, and some main academic publishers including ACM, IEEE, Springer, Elsevier, and Wiley. We collected papers that have either "concurrency bug", "concurrent program", "multithreaded program", "data race", "atomicity violations", "order violations", or "deadlock" keywords in their titles, abstracts, or keywords. Secondly, we included papers published during the period from 1993 to 2016 and excluded those marginal contributions and flawed designs. After a review of the results, we manually screened and added some articles with fundamental contributions even though they are not available in the above databases. Finally, we collected over 100 papers within the scope of our survey. All of these papers cover quite a number of research works published at leading conferences or journals, such as ICSE, OSDI, SOSP, PLDI, ASPLOS, HPCA, ISCA, TSE, TOPLAS, TOCS, etc. These papers are classified into four categories as follows:

(1) Automated concurrency bug exposing: Studies concerning different technologies to make concurrency bugs manifest and reproduced effectively and efficiently.
(2) Automated concurrency bug detection: Studies concerning exploring effective ways to identify the failure-inducing thread execution scheduling and even the root cause.
(3) Automated concurrency bug avoidance: Studies concerning how to tolerate exiting bugs by preventing concurrency programs from failure.
(4) Automated concurrency bug fixing: Studies concerning exploring different ways of fixing concurrency bugs.

The four categories of studies are defined according to their goals. Concurrency bug exposing aims to manifest the bug-triggering interleaving efficiently, which is the precondition of identifying the bug's root cause and even fixing it. An effective bug-exposing approach would facilitate detecting, avoiding, and fixing concurrency bugs. Concurrency bug detection focuses more on localizing a bug and identifying the root cause. It provides the diagnosis of bugs to bug avoidance and fixing methods. Concurrency bug avoidance and fixing would both help generate correct results of multi-threaded programs. The main difference is that concurrency bug fixing is more useful for repairing program bugs than concurrency bug avoidance.

Figure 1 shows the category distribution of the current state of research in concurrency bug exposing, detection, avoidance, and fixing. Note that the categories are not exclusive: that is, a paper may cover topics in multiple categories. We manually assign each paper to one or more categories based on its main objective and contributions. Thus, our classification is subjective and some papers may be classified into a different category by other researchers. However, we believe that in general, the distribution shown in Fig. 1 represents a fairly good indication of the topic distribution of the current state of research in addressing concurrency bug problems. Figure 2 illustrates the number of articles in this area from 1993 to 2016 published on conference, on journal and the sum of them, respectively.

1.2 Concurrency bugs classifications

Tan et al. (2014) conducted a comprehensive study of 2060 real-world bugs, which are divided into four types: memory bugs, semantic bugs, concurrency bugs, and performance bugs. Among them, we are particularly interested in concurrency bugs, which can be further classified into the following four categories: data races, atomicity violations, order violations, and deadlocks.

A data race occurs when there are two concurrent accesses from different threads to the same memory location and at least one of them performs a write operation. Figure 3 shows a data race bug example in fast Fourier transform (FFT) transformation (Deng et al. 2013). In this example, the programmer plans to execute statement S2 in Thread 2 before executing

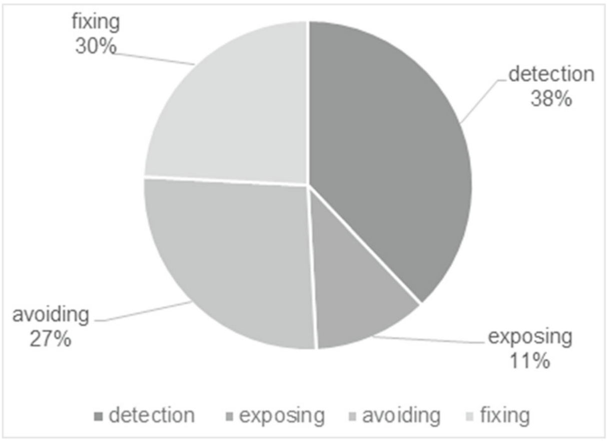


Fig. 1 Classification of research topics on concurrency bugs

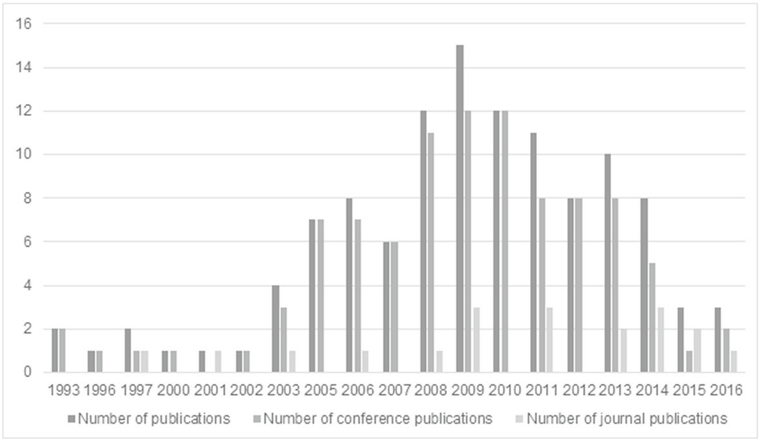


Fig. 2 Number of publications per year

statement S1 in Thread 1. However, this is not guaranteed in the current block of codes, since the proper synchronization mechanism to guarantee the desired execution order is missed. As a result, in some cases S1 may execute before S2 and thus lead to an incorrect solution.

An atomicity violation occurs when two program blocks from two threads interleave unserializably, which violates the expected atomicity of one or both blocks. Figure 4 shows an atomicity violation extracted from MySQL (Park et al. 2012). The read access S3 of the variable log_type in Thread 2 may interleave between the two write accesses, S1 and S2, in Thread 1. The programmer did not protect S1 and S2 in an atomic section. The bad interleaving leads S3 to using the incorrect definition by S1, and thus produces a wrong answer or a program crash.

An order violation occurs when an instruction P is expected to execute before/after another instruction Q, but actually executes after/before Q ascribe to the lack of synchronization. Lu et al. (2008) first presented the order violation as a category of concurrency bugs. For example, in Fig. 5 (Shi et al. 2010), the programmer assumed that statement S3 in Thread 2 should always happen before S2 in Thread 1. A crash would happen once an order violation bug occurs, e.g., S2 occasionally comes before S3.

Deadlock can usually be divided into two types: resource deadlocks and communication deadlocks, respectively (Joshi et al. 2010). A resource deadlock occurs when at least two

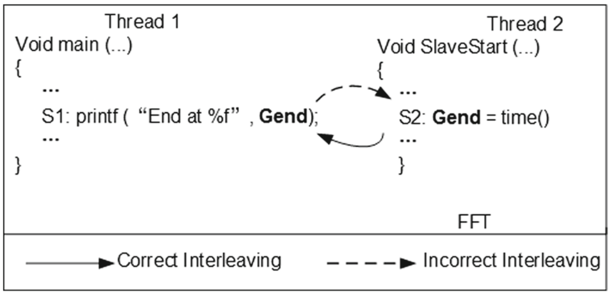

Thread 1
```
Void main (...)
{
    ...
    S1: printf ( "End at %f" , Gend);
    ...
}
```
Thread 2
```
Void SlaveStart (...)
{
    ...
    S2: Gend = time()
    ...
}
```

Fig. 3 A data race bug in FFT (Deng et al. 2013)

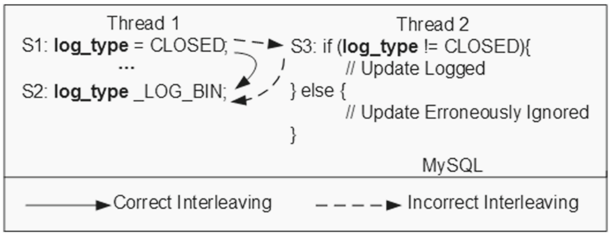


Fig. 4 An atomicity violation bug in MySQL (Park et al. 2012)

threads wait to acquire a synchronization object that is held by another thread related to one of the competing threads, resulting in a circular waiting cycle between them. A communication deadlock is communication deadlock. It is usually caused by a set of threads blocks waiting for a lock held by one thread within the same set. It is rooted in the misuse of condition variables in threads blocks or abnormal mutual effect between deadlocks and condition variables. Figure 6 depicts a real-world deadlock bug in HawkNL (Zhang et al. 2013). As we can see, both Thread 1 and Thread 2 could acquire *nlock* and *slock*. Thread 1 acquires *nlock* first and Thread 2 acquires *slock* first, and each of them is waiting for the other thread to release the held lock. This situation will lead to a deadlock.

1.3 The features of concurrency bugs

Most concurrency bugs exist in concurrent programs of operating systems. Thus, these bugs are more likely to cause program hangs or crashes (Tan et al. 2014), which may cause significant and irreparable losses. However, the space of execution interleaving between concurrent programs expands exponentially with the increasing of the number of threads and program size. Analyzing concurrent programs becomes computationally prohibitive when the program size is significantly large. The execution of concurrent programs are of high indeterminacy even if the inputs are identical. Thus, in practice, programmers are quite vulnerable to concurrency bugs due to the highly intractable feature of concurrent programs. This also poses great challenges for us to detect, expose, avoid, and fix concurrency bugs.

Another challenge of addressing concurrent bug problems is that it is difficult to be reproduced because of the large state space. Most concurrency bugs could not be reproduced within an acceptable probability. In the worst case, the reproducibility rate could be lower than 10%. The low reproduction rate makes concurrency bugs difficult to detect and expose. Concurrency bugs are usually hidden in certain rare interleavings. It is infeasible to traverse

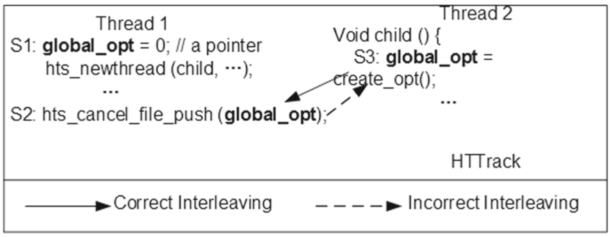
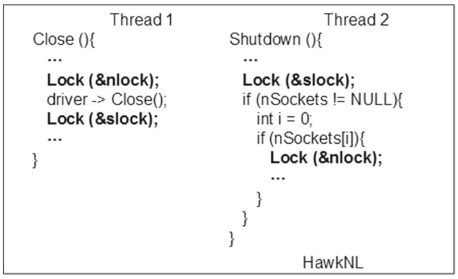


Fig. 5 An order violation bug in HTTrack (Shi et al. 2010)



```
        Thread 1                     Thread 2
Close (){                    Shutdown (){
  ...                          ...
  Lock (&nlock);               Lock (&slock);
  driver -> Close();           if (nSockets != NULL){
  Lock (&slock);                 int i = 0;
  ...                            if (nSockets[i]){
}                                  Lock (&nlock);
                                   ...
                                 }
                               }
                             }
                                        HawkNL
```

Fig. 6 A deadlock bug in HawkNL (Zhang et al. 2013)

all possible interleavings. Which part of interleavings should be checked and how to check are key issues to be addressed in concurrency-bug detection and exposing.

Even though concurrency bugs can be detected, they are hard to fix. The fixing process may take a long time and the generated patches may easily lead to new errors (Yin et al. 2011). Patches usually do not really fix bugs but decrease the probability of their manifestation.

Detecting, exposing, and fixing concurrency bugs are confronted with several key challenges as follows:

(1) Challenges of concurrency-bug exposing: The huge interleaving space is the biggest challenge against effective concurrency-bug exposure. High coverage criteria is needed for exposing different buggy interleaving patterns. Different inputs may lead to different execution paths and results for the same concurrent program. Therefore, significant research efforts have been devoted to the study of input generation in the literature.

(2) Challenges of concurrency-bug detection: Similar to concurrency-bug exposing, huge interleaving space also makes it cumbersome to effectively detect concurrency bugs. Both static and dynamic detection techniques produce false positives and false negatives. Another important aspect of evaluating a concurrency bug detection technique is the constraint of limiting the overhead within a reasonable level.

(3) Challenges of concurrency-bug fixing: A vital prerequisite for concurrency-bug fixing is to identify the root causes of the bugs. Existing work either develops new detection methods or leverages existing tools to identify the root causes. In addition to generating correct patches for concurrency bugs, techniques for fixing concurrency bugs should not introduce new bugs into the original program and reduce program performance and code readability significantly.

Deng et al. (2015) conducted a survey in 2015 and discussed the existing progress on fixing, preventing, and recovering from concurrency bugs. They mainly review their work on fixing, preventing, and recovering from concurrency bugs and make a comparative analysis on three techniques: CFix, AI, and ConAir. Based on this, we further collect the latest research achievements related to the study of exposing, detecting, avoiding and fixing concurrency bugs. Compared with Deng et al.'s work, we make a more systematic and integrated review of different aspects of concurrency bugs. For each aspect, we review the key issues, challenges, and the current state of the art in order to make a comprehensive review of this hot research issue.

The remainder of this article is organized as follows. We firstly discuss the techniques and research achievements in exposing concurrency bugs in Section 2, and move to the topic of concurrency bugs detection in Section 3. Sections 4 and 5 focus on avoiding concurrency bugs and fixing concurrency bugs, respectively. We conclude and present the benchmarks that are the most actively used in Section 6. In Section 7, we conclude our survey with future directions.

2 Automated concurrency bug exposing

Concurrency bug exposing is important because it is the prerequisite for concurrency bug detection, avoidance, and fixing. For most concurrency bugs, the thread interleavings can expose them rarely without any perturbation during the execution. Stress-testing is a common and traditional practice to expose concurrency bugs. Nevertheless, stress-testing is evaluated to be inadequate because it tends to test similar thread interleavings repeatedly over an input. As a result, it is almost impossible to expose a rare buggy interleaving through stress-testing. Concurrency bugs are exposed with a low rate mainly because: (1) The interleaving space is huge and grows exponentially with the size of the code. (2) Concurrency bugs are always concealed in certain rare and special memory access interleavings. (3) Concurrent programs will be executed along with the same interleaving pattern without extra scheduling. To overcome these problems, researchers have done lots of work and have presented various solutions. A good bug-exposing technique must be effective, efficient, and reproducible, so that it will be accepted by more developers.

Figure 7 shows the process of concurrency bug exposing. Concurrent programs are firstly analyzed to identify suspicious interleavings. Then a controlled testing is conducted with different strategies appended to the programs in order to expose bugs. The concurrency bug-exposing techniques come in three types and will be introduced separately in the following three sections.

2.1 Random delay disturbance

This kind of technique inserts random disturbance when concurrent programs are accessing shared memory and synchronizing, which can effectively increase the probability of exposing all kinds of concurrency bugs. Essentially, with the random delay disturbance, the probability of triggering the rare execution interleaving rises.

Bron et al. (2005) employed random delay disturbance to design a new coverage model called a synchronization model to control the executions of different threads. Coverage analysis is useful for exposing techniques. In order to check whether all synchronization statements are tested in concurrent programs, Bron et al. added random delay disturbance in code and force a thread to execute by blocking another one. An input and an interleaving, which can trigger a concurrent bug, are essential for exposing it. Stress-testing is known as

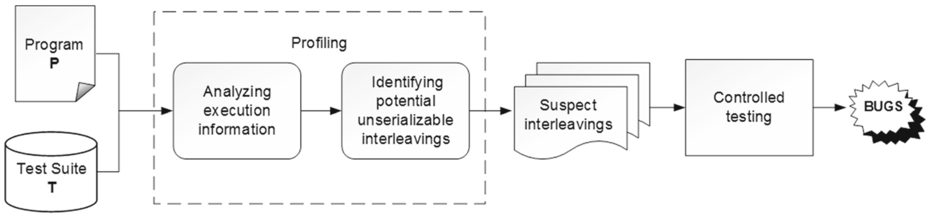

Program
P
Test Suite
T
Profiling
Analyzing execution information
Identifying potential unserializable interleavings
Suspect interleavings
Controlled testing
BUGS

Fig. 7 The generalized framework of exposing

a common practice to expose concurrency bugs but empirical evidence clearly demonstrates that this form of testing is inadequate. To compensate, several previous works (Sen 2008; Musuvathi et al. 2008) have focused on improving stress-testing. However, these works select only one thread to execute at a time, which causes much overhead during the testing runs.

To address the limitation of stress-testing, researchers are devoted to studying more efficient testing methods. CTrigger (Park et al. 2009; Lu et al. 2011b) employs random delay disturbance and allows each test run to use multiple processors so that it can effectively and efficiently expose atomicity violations. It firstly conducts a study of some program executions and identifies unserializable interleavings. Then it applies a probability to each feasible interleaving and ranks them in order to make low-probability interleavings expose atomicity violations. CTrigger inserts synchronization manually at a few undetermined points of programs. Similar to CTrigger, Chew (2009) proposes a system to detect, prevent, and expose atomicity violations. For exposing bugs, the system injects delays randomly between memory accesses to make bugs easier to be expose. This technique exposes atomicity violations online, but CTrigger attempts to expose atomicity violations offline. Due to the different goals, Chew puts more efforts on reducing performance overhead. To reduce excessive overhead of previous techniques, Kivati (Chew and Lie 2010) is proposed to find and prevent bugs with low overhead. The remote thread was suspended by Kivati so as to protect the atomicity region. Kivati is able to detect quickly and prevent atomicity violations on commodity hardware with a low overhead.

2.2 Thread scheduling/switch

The thread scheduling/switch technique controls the scheduling mechanism directly or indirectly. Before or after a thread accessing shared memory or doing synchronization control, the technique can suspend the current thread and switch to execute another one.

Other than inserting random disturbance, Bron et al. (2005) also leverage thread scheduling to make more combination forms of thread interleavings. CHESS (Musuvathi et al. 2008) is a tool that can be used to find and reproduce concurrency bugs. It is like the stateless model checking and detects both safety and liveness violations. CHESS controls thread scheduling with its scheduler to execute a large number of code and can reproduce crashing bugs persistently even if no one knows the root cause. Musuvathi et al. (Musuvathi and Qadeer 2007) propose an iterative context-bounding algorithm and implement it in CHESS. This algorithm studies the relationship between context switches in an execution of multithreaded programs and the efficiency when testing multithreaded programs. A context switch means a thread stops executing temporarily and another thread starts. They further discuss preempting context switch, which occurs when a thread stops execution at an arbitrary point by the scheduler. This algorithm makes a limitation to the number of context switches but leaves the depth of the execution unlimited. Their experimental results show that under the limit of two preempting context switches, the iterative context-bounding algorithm found nine out of 16 bugs. Similar to CHESS on reducing the test space with a small number of preemptions, Maple (Yu et al. 2012) also bounds the number of interthread memory dependencies to two and increases the interleaving coverage by memorizing tested interleaving and exposing untested interleavings. It develops an online technique to predict which untested interleaving can be exposed for a given test input and then it controls thread scheduling while program executing so that the predicted untested interleavings are exposed.

In contrast, PCT (probabilistic concurrency testing) (Burckhardt et al. 2010) can introduce an arbitrary number of preemptions even if the bug depth is small. PCT is a randomized algorithm that is used to test concurrent programs as a scheduler. The key idea of PCT is to find a concurrency bug regarding the characterization of the depth of the bug as the minimum number of context scheduling. This would raise programmers' confidence when testing concurrent programs.

2.3 Fuzzing

Fuzzing firstly detects potential concurrency bugs by bug detectors and then controls thread scheduling and execution according to the bug reports so that the program may execute specific interleaving patterns to expose concurrency bugs. Most of the fuzzing methods get bug-manifestation features based on static analysis on real-world concurrency bugs. Due to the ability of exposing concurrency bugs purposefully, fuzzing it is more efficient than other methods.

With the motivation of replaying data races with high efficiency and low overhead, RaceFuzzer (Sen 2008) can find and reproduce bugs in concurrent programs. It firstly detects suspicious races by a detection technique, and then makes a randomized thread scheduler to control thread execution in order to trigger a real data race from those potential ones detected before. RaceFuzzer can pick up real data races, which may cause program exceptions from potential ones and easily reproduce them. Unfortunately, the bug-exposing capability of RaceFuzzer relies a lot on the underlying data race detectors. Many bugs would be ignored due to a bad coverage of the underlying detectors.

For deadlock exposing, DeadlockFuzzer (Joshi et al. 2009) detects real deadlocks in multithreaded programs. Firstly, DeadlockFuzzer finds suspicious deadlocks by a dynamic technique. Next, it controls thread scheduling with a randomized scheduler to replay the suspicious deadlocks so that these deadlocks can be exposed with high probability. Cai et al. (2014) experimentally show that DeadlockFuzzer is unable to confirm a real deadlock with high probability. To address this problem, they present ConLock (Cai et al. 2014) to dynamically check deadlocks using constraint-based approaches. They firstly study a given potential deadlock to get a set of constraints of thread scheduling. These constraints describe the rules that the order of acquiring or releasing locks of the pair of threads involved in that given deadlock. After that, they run the program within the constraints in order to create the deadlock. If the deadlock cannot be created, ConLock reports this as false positive. The validation process proves that ConLock reports all 11 deadlocks of real-world programs with high confirmation probability within the range of 71 to 100%.

2.4 Conclusions of concurrency bug exposing methods

Stress-testing used to be treated as a common method for concurrency bug exposing. However, stress-testing has many limitations such as inadequacy and inefficacy, which motivates some alternative techniques with different strategies. To conclude, inserting random delay disturbance indirectly affects the scheduling of threads, while thread scheduling/switch methods directly control thread scheduling. Random delay disturbance can be used to analyze synchronization coverage, which is a very important measurement criterion for exposing. However, inserting random delay disturbance may cause significant performance overhead. Thread scheduling/switch technique controls multithreaded programs to run like a scheduler. This technique is targeted to expose concurrency bugs, which is different from random delay disturbance technique. However, none of the above approaches consider the impact on thread scheduling. Different from them, fuzzing methods consciously control thread scheduling in order to increase the likelihood of exposing concurrency bugs, but rely on the detection results of other tools. In addition, fuzzing has advantages in deadlock exposing.

3 Automated concurrency bug detection

Identifying concurrency bugs is time-consuming and usually requires extensive debugging experience for programmers. Automated concurrency bug-detection techniques could help programmers to find bugs timely and accurately. However, the detection results might be misleading, which falls into the following two categories: false positives and false negatives. A false positive is reported when a bug-free pattern is treated as a buggy one while a false negative ignores a really buggy pattern. To reduce both false-positive and false-negative rate, many approaches have been proposed.

In this section, we summarize the main contributions to concurrency bug detection in the literature. Figure 8 depicts the whole process of concurrency bug detection. Given the concurrent program *P* and test suit *T*, different methods can be applied to detect concurrency bugs. Based on whether programs are executed or not, we classify all detection methods into three categories: static, dynamic, and hybrid. Each of these will be discussed in detail in the rest of this section.

3.1 Static methods

Static methods detect concurrency bugs by analyzing suspicious code paths without executing programs. This type of method can find bugs and test the paths, which is hardly reached during programs running by off-line analysis. As a result, static methods are more effective than dynamic ones (Engler and Ashcraft 2003).

Concurrency bugs are scheduler-dependent, which are difficult to be found by compile-time checks like traditional testing techniques. Flanagan and Freund (2000) present a static race detection analysis for concurrent Java programs. This analysis system proceeds from the race-avoidance rules, which is lock-based synchronization. Then the system tests whether a lock is held correctly no matter when a shared variable is accessed, but their system requires programmers to write additional type annotations, which may cause extra overhead. Based on this, Abadi et al. (2006) perform a static race-detection analysis for large-scale concurrent Java programs. The analysis is type-based, without the limitation of the test coverage concerns. The analysis is feasible for thread-local classes and those classes that have internal synchronization or require client-side synchronization. They evaluated the effectiveness of the technique on 4000 lines of Java code and detected a number of race conditions in the standard Java libraries and some other test programs. Engler presented

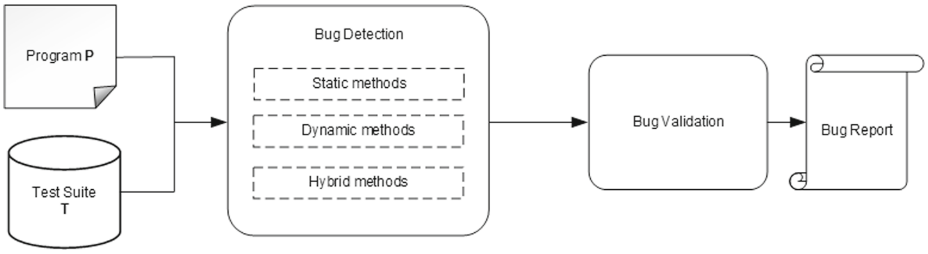


Fig. 8 The whole process of concurrency bug detection

RacerX (Engler and Ashcraft 2003), which can detect race conditions and deadlock without requiring annotations. RacerX effectively finds race conditions and deadlocks by concurrent inter-procedural program analysis and flow-sensitive analysis in large systems. It finds 16 bugs in two operating systems with totally over 2.3 MLOC. Though RacerX can handle large programs, only a relative small amount of bugs (high false-negative rate) can be detected.

To improve RacerX, Naik et al. (2006) propose a novel static technique for Java program race detection. Their algorithm is context-sensitive but not flow-sensitive. The four stages of the algorithm, which are reachable pairs, aliasing pairs, escaping pairs, and unlocked pairs, sequentially refine the race-related memory accesses. The key approach of their algorithm is a form of context sensitivity, which makes this technique scalable to be applied in large, widely used programs. This technique finds more bugs than all previous static race detection techniques including RacerX. In their largest benchmark, derby, the technique reports 1018 races revealing 319 distinct bugs.

As an improvement of RacerX, Naik and Aiken (2007) then present a new algorithm. This algorithm addresses RacerX's limitation in proving race freedom. They consider that the core to prove race freedom is to show that if two locks are distinct, then the memory locations they guard are also distinct. They present *conditional must not aliasing*: under the assumption that two objects are not aliased, prove that two other objects are not aliased. They use *conditional must not aliasing* to analyze multi-threaded programs and detect races statically. Vaziri et al. (2006) think that most approaches are code-centric, as a result, the approaches do not consider the problems caused by inconsistency lock mechanism when accessing shared data throughout the program. They present a data-centric approach (Vaziri et al. 2006) and present 11 suspicious interleaving patterns that are considered as the new definition of data races. Besides, they statically analyze which part of the code must be modified to avoid data races on condition of the new definition.

All methods mentioned above are applied to Java programs. There is also another method applied to C programs. Voung et al. (2007) present RELAY, a static and scalable technique to detect data race in the Linux kernel. Unlike RacerX, which uses a top-down approach to compute the lock-sets at each program point, RELAY performs a bottom-up context-sensitive analysis to detect data races. The key idea of RELAY is relative lockset, which helps them analyze the behavior of a function independent of the calling context. RELAY found 53 races for a 4.5 million lines of C code. However, its false-positive rate is more than 70%.

3.2 Dynamic methods

At present, the main method to detect concurrency bugs is the dynamic method. The advantage of dynamic methods is that they only check observed feasible execution paths and can accurately judge the values of variables and threads interleaving modes. As a result, dynamic methods are more flexible for detecting different problems, such as plagiarism detection (Tian et al. 2017). Nevertheless, the disadvantage of dynamic methods is the heavy overhead and thus cannot be implemented in those programs that are sensitive to speed.

To locate the faults in concurrent programs, Savage et al. (1997) present Eraser, a dynamic tool using lockset algorithm that can detect data races. This is the first dynamic race-detection tool used in multithreaded production servers. They use Eraser for lock-based synchronization and to make sure shared-memory accesses confirms the programming policy, which can protect programs from data races. Eraser has the advantage of checking unannotated programs, but it may fail to detect certain errors because of insufficient test coverage. Smaragdakis et al. (2012) introduce a new relation called causally precedes (CP), which is the generalization of the happens-before algorithm and can observe more races and has no loss of accuracy and completeness. CP is superior to past detection approaches on soundness and of polynomial complexity.

As a hybridization of both lockset and happens-before algorithms, Choi et al. (2002) present an approach to detect data races dynamically for object-oriented programs. Their experimental results on benchmarks $mtrt$, tsp, $sor2$, $elevator$, and $hedc$ show that this approach significantly outperforms existing techniques in terms of detection efficiency improvement and overhead reduction. Eraser enforces a unique lock to protect every shared variable, however, this approach does not take the same strategy as Eraser and it has lower overhead but higher precision. Unlike Eraser and Choi's approach, Yu et al. (2005) propose a practical and run-time race detection tool called RaceTrack. It executes instrumentation at the virtual machine level, which is designed for object-oriented programs and implemented in the virtual machine of Microsoft's Common Language Runtime. RaceTrack first tracks the program execution traces based on the instrumentation information. Then it reports suspicious patterns of memory accesses once they are observed. It also uses a hybrid detection algorithm to monitor memory-shared accesses in order to improve the accuracy.

Different from any lockset and happens-before algorithms and without requiring a priori program annotation, Xu et al. (2005) design a Serializability Violation Detector (SVD) to protect buggy concurrent programs from errors using backward error recovery (BER) and to provide errors root causes for programs debugging. Instead of requiring a priori annotations, SVD enables a posteriori examination, which can be readily applied to large and important programs. Flanagan et al. argue SVD enforcing Strict 2-Phase Locking to ensure serializability is sufficient but not necessary, and thus they take the lead in performing a sound and complete dynamic analysis (Flanagan et al. 2008) for atomicity violations. Given a program, the analysis explores the accurate dependencies between memory accesses in certain code regions. If there are accesses that are not conflict-serializable, the analysis provides a detection report of error messages. Similar to SVD, ToleRace (Ratanaworabhan et al. 2009) is also distinct from the lockset and happens-before algorithms, but it only focuses on asymmetric races. ToleRace is able to detect and tolerate races by a transaction-like approach and reduces the overhead of dynamic race detection to 6.4% on average.

Most previous work only targets one specific type of concurrency bug. For example, Velodrome (Flanagan et al. 2008) is only designed for checking atomicity violations. Different from those works, Jin et al. (2010) propose Cooperative Crug Isolation (CCI), a technique designed for a wide variety of concurrency bugs. CCI has low overhead and satisfactory scalability on concurrency bug detection and root-cause analysis. It collects the information of predicates related to memory. The random sampling strategies of CCI lower the run-time overhead. Sampling information is processed by statistical models to effectively detect concurrency bugs and obtain their root causes.

As one of the closest studies to CCI, Falcon (Park et al. 2010) studies the constructs corresponding directly to faults and also takes the ranking scheme. Falcon is a dynamic fault localization tool which can present faulty data-access patterns and also detect different types of concurrency bugs. It uses pattern-based analysis method to capture both order violations and atomicity violations. It first works with a set of conflicting interleaving patterns that induce order violation and a set of unserializable interleaving patterns that induce atomicity violation, then monitors program execution to detect if actual interleavings match the two kinds of patterns and finally it uses pass/fail statistics to compute a suspiciousness value for all occurring patterns. Comparing to CCI, Falcon may find the real bugs more effectively. UNICORN (Park et al. 2012) modified Falcon by monitoring pairs of memory access and then combining those pairs into patterns as the ranking units implemented in both Java and C++. Unlike Falcon, UNICORN can detect both single-variable and multi-variable violations, so that the most important classes of non-deadlock concurrency bugs can be covered by UNICORN.

The detection tools mentioned above rely on specific bug patterns, but Recon (Lucia et al. 2011) is not pattern-based and generally used for handling a variety of single- and multi-variable errors. Recon has the advantage of not only bug detection, but also reporting a short fragment of failure-inducing execution schedules, which can help programmers understand the bugs better.

Some methods can not only detect concurrency bugs but also tolerate bugs during production runs. Zhang et al. (2016) propose an innovative program invariant called Anticipating Invariant (AI) to detect most kinds of concurrency bugs. They use AI to generate a software-only system that can detect buggy thread interleavings right before the time when bugs can be avoided with low-overhead. AI can also be used to tolerate many concurrency bugs, expose order violations, and generate emergency patches.

Improper uses of locks may lead to deadlocks and prevent programs from further progress, so some detection techniques for deadlocks are proposed exclusively. Cai et al. (2012) present MagicFuzzer for deadlock detection. After executing a tested program, MagicFuzzer produces a log containing lock dependency information. Next, an algorithm called Magiclock identifies hidden deadlock cycles from the log. Finally, MagicFuzzer tries to trigger the potential deadlock cycles and reports it if one deadlock cycle occurs. To improve the efficiency and scalability of this technique, MagicFuzzer lessens lock dependencies for cycle detection. Later, based on this work, they improved the algorithm and presented Magiclock (Cai and Chan 2014), which is considered a more generalized algorithm. Cai et al. (2016) propose ConLock$^+$ to address the limitation in scalability and false positives of existing deadlock detectors. Since a deadlock always leads to a cycle that at least two threads are waiting for each other, this technique generates a set of constraints for each cycle by analysis, and the involved execution in order to get a set of scheduling constraints that may cause deadlock. Then, ConLock$^+$ is implemented to verify whether a cycle exists and whether a deadlock can be triggered. Once the deadlock is triggered, ConLock$^+$ does a follow-up analysis to identify the involved memory accesses. This technique has been implemented on 16 real deadlocks in several real-world programs, and has shown good effectiveness and efficiency.

Unlike other work that focuses on bugs caused by buggy interleavings, Zhang et al. (2010) focus on bugs that can cause program crashes and built ConMem. These bugs are caused by incorrect thread interleavings leading to memory problems. ConMem detects any erroneous interleavings that can lead to program crashes by monitoring program execution, analyzing memory accesses, and synchronizations. The experimental results show that ConMem detects eight out of nine real-world severe concurrency bugs with much lower false-positive rates. Asymmetric race is a special class of race conditions. Especially, in a multithread program, one thread correctly runs with the lock of a given shared variable which is incorrectly accessed by another thread.

3.3 Hybrid methods

Besides the above two main types of concurrency bug detectors, there are still some other methods that either combine static and dynamic methods together or combine one of the two methods with other techniques. Further, we include papers using formal methods for concurrency bug detection and make an analysis of these papers in two categories: symbolic execution and model checking.

3.3.1 Combination of static and dynamic methods

The combinations of static and dynamic methods leverage advantages from those combined ones to obtain more powerful detection ability.

Lu et al. (2007) first analyzed the limitations of previous approaches and then found that most previous techniques that detect concurrency bugs need specific synchronization semantics, which is difficult to recognize without prior knowledge and may lead to spurious bug reports. Although SVD is the first tool for detecting atomicity violation, it can only be applied to a limited subset of atomicity violation bugs. As a result, Lu et al. (2007) firstly propose an atomicity violation detection tool called AVIO, combining static approaches and dynamic approaches together. They mention Access-Interleaving invariants (AI invariants), which represent those parts of code that are expected to execute atomically. AI invariants are extracted through a large number of correct runs. Once certain memory access interleavings violate those invariants at run time, concurrency bugs are detected. Similarly, Shi et al. (2010) also extract invariants from training runs and then dynamically detect violations. They present a new concept named definition-use invariants (DefUse invariants) (Shi et al. 2010), which describe the intrinsic relationships between definitions and uses for multi-threaded and single-threaded programs. Local/Remote (LR) invariants, follower invariants, and definition set (DSet) invariants are three types of DefUse invariants. The detection process includes extraction of DefUse invariants and detection of a variety of program bugs leveraging DefUse invariants. Unlike previous work, this tool focuses on different invariants and data flow, and it is the first method to detect both concurrency and sequential bugs. Due to the fundamental limitations of the existing work, such as non-negligible false negatives and false positives and user unfriendliness, a consequence-oriented approach is presented by Zhang and his colleagues. Zhang et al. (2011) summarize a bug's lifecycle into three stages: (1) triggering, (2) propagation, and (3) failure. Different from traditional techniques that put much effort on stage (1), the proposed tool takes a consequence-oriented approach, which detects concurrency bugs from the back forward. That is, the approach first statically finds the buggy code region in a program binary, and then at run time it monitors the execution of a tested program and identifies the memory accesses that can lead to software failure. This approach effectively improves the bug-detection coverage and accuracy.

Kasikci et al. (2013) find that static and dynamic detection tools have their own drawbacks in the performance of false positives, false negatives, and overhead. Both static and dynamic methods have low accuracy and are not very practical. Kasikci et al. present RaceMob (Kasikci et al. 2013), a data race detector with good accuracy. RaceMob first statically detects suspicious data races, then it dynamically validates if all detected races are true. The combination of static and dynamic detection helps to guarantee low runtime overhead and high accuracy. Finally, in the list provided by RaceMob, data races with different severity are reported to programmers for efficient debugging. Different strategies are taken in dynamic phase by Deng et al. (2013). These strategies are applied to a set of inputs to reduce effort of detecting concurrency bugs, which can therefore improve the detecting performance. They conducted a study and found that existing concurrency bug detectors are not applicable for many inputs. In dynamic phase, they present Concurrent Function Pair (CFP), a new interleaving-coverage metric, and leverage it to detect concurrency bugs. The evaluation shows that CFP-guided concurrency-bug detection is more efficient in improving bug-detection efficiency than previous works.

Also, innovative and practical tools are presented. Aiming at semantic and concurrency bugs, MUVI (Lu et al. 2007) focuses on detecting inconsistent updates and multi-variable concurrency bugs. MUVI combines static program analysis and data mining techniques. It is the first tool to identify multi-variable access correlations that can be used to detect multi-variable inconsistent update bugs. They also extend lockset and happens-before detection methods and present multi-variable data race detection methods as a supplement of existing techniques. In their subsequent work (Lu et al. 2011a), they propose two program invariants by studying the characteristics of real-world concurrency bugs and implement AVIO and MUVI as two feasible tools. The evaluation results show that both AVIO and MUVI are effective tools with fewer false positives and higher accuracy. Portend$^+$, a tool presented by Kasikci et al. (2015), not only detects data races but also analyzes the potential consequences in order to judge the severeness of each bug. In their definition, data races come in four main types: specification violated, output differs, k-witness harmless, and single ordering. They combine multipath and multischedule analysis with symbolic program-output comparison to classify data races according to their severity. Portend$^+$ is the first technique to classify data races and the classification helps programmers understand bugs better.

3.3.2 Formal methods

In computer science, formal methods are a kind of mathematically based techniques for specifying and verifying complex software and hardware systems (Clarke and Wing 1996). Applying formal methods can improve the reliability and robustness of a system (Michael 1997). For concurrent programs, formal methods, such as symbolic execution methods and model checking methods, can reduce the complexity of the program analysis. These methods use different strategies from static and dynamic methods and we will make a detailed discussion on these.

(1) Symbolic execution

Symbolic execution assumes symbolic values for inputs rather than actual inputs as normal execution of the program. It has already been applied for systematic testing sequential programs (Cadar et al. 2008; Godefroid et al. 2005; Sen et al. 2005; Tillmann and Halleux 2008). It is also useful for testing concurrent programs. Many techniques use symbolic execution to prune redundant executions in concurrent programs, which makes concurrency bug detection much easier.

Predictive analysis detects concurrency bugs during run-time by monitoring execution trace. Some predictive bug detection methods that are based on satisfiability modulo theory (SMT) solvers (Wang et al. 2010; Wang et al. 2011) have been presented. These methods utilize the source code and execution trace to build a symbolic predictive model. Then a SMT solver is employed to check the potential interleavings for concurrency bugs. Kundu et al. (2010) do not directly detect bugs in concurrent programs, and present CONTESSA to improve the coverage of testing using SMT solvers. CONTESSA leverages symbolic analysis to explore thread interleavings. Symbolic analysis improves the efficiency of program testing because it avoids enumeration of interleavings. Despite these bug detectors, another SMT-based symbolic method (Said et al. 2011) focuses on helping programmers understand how a data race can happen during program execution. The method uses symbolic analysis to search for concrete thread schedules, which can deterministically trigger the data races, among alternative interleavings of a given execution trace. The concrete thread schedules can help programmers debug.

Rungta et al. (2009) find that exhaustive search techniques such as symbolic execution are insufficient to detect bugs in concurrent programs. They present an abstraction-guided symbolic execution technique that can quickly detect concurrency bugs. The techniques firstly identify an abstract system that contains a set of suspicious program locations that may lead to an error state. Then the symbolic execution is guided along these locations to generate an execution path to the error state. Wang et al. (2017) propose a new symbolic execution approach to detect faults without exploring redundant paths. This approach resolves the path explosion problem brought by traditional symbolic execution and shows excellent fault-detection ability.

(2) Model checking

Model checking is an effective way to analyze and test concurrent programs based on state-space exploration algorithms. State-space exploration techniques usually explore a directed graph, called the state space, which represents the combined behaviors of all concurrent components in a system (Godefroid 1997). Important properties such as deadlock and liveness can be verified automatically using graph based model-checking algorithms. In many cases, model checking is capable to find concurrency bugs even for complex systems (Clarke et al. 1995; Boigelot and Godefroid 1996).

Among model checking methods, context-bounded analysis has made great contribution to concurrency bug detection. Bounding the number of context switches reduces the complexity for analyzing concurrent programs. Qadeer and Rehof (2005) first propose a theory of context-bounded model checking for concurrent programs. Their technique can find any bugs within a bounded number of context switches. Similarly, Lal and Reps (2009) present a method for reducing concurrent analysis under a context bound. Unlike Qadeer and Rehof's technique, their method can conduct context-bounded analysis with any given context bound and with different program abstractions. Lahiri et al. (2009) use SMT solvers and apply context-bounded analysis to concurrent C programs. This work eliminates the complexity of Lal and Reps's work (Lal and Reps 2009). It also finds a new bug in a set of real-world programs during the evaluation process. Besides, some approaches (Rabinovitz and Grumberg 2005) relies on bounding the number of contexts to address the complexity and scalability problems of concurrent program analysis.

Besides context-bounded analysis, another way to solve the state explosion problem is using partial order techniques. Wang et al. (2008) present a symbolic dynamic partial order reduction (POR) method for model checking concurrent programs. This POR method detects data races with less time compared to an existing method (Kahlon et al. 2006). Besides, it can also remove all redundant interleavings in a two-thread concurrent program.

3.4 Conclusions of concurrency bug detection methods

In conclusion, static detection methods attempt to detect concurrency bugs by reasoning about source code. Most of these methods employ compile-time analysis on the source code and then report all potential races that could occur in any possible program execution. Static methods cannot avoid excessive false positives because the compile-time analysis cannot make a precise estimation of the possible bugs. Besides, static methods have bad scalability because it is hard to analyze a large program entirely. For dynamic detection methods, they discover bugs by monitoring particular execution of a program. Using dynamic methods must execute the tested programs and record the history of its memory accesses and synchronization operations for subsequent analysis. Since the history is in fact a feasible execution, dynamic methods report lower false-positive rates than static ones. The happens-before and lockset algorithms are very important among dynamic methods. Those methods based on happens-before relation are more precise and those lockset-based methods are more efficient. Therefore, there are some dynamic detectors that combine happens-before and lockset together to improve the efficiency and accuracy of detection results simultaneously.

Nevertheless, given the restriction that not all possible execution paths are executed and recorded, dynamic methods are not sound and can hardly prove the nonexistence of concurrency bugs. As a result, many hybrid methods are proposed for better detection efficiency and accuracy, also for a widely variety of concurrency bugs. Those hybrid methods may take advantage of each individual method and show better performance.

Formal methods are quite different from static, dynamic, and other kinds of detection techniques, and they are promising alternatives to the above-mentioned techniques. Formal methods have the advantage of improving the coverage of testing. However, they have limitations on scalability due to the well-known state-explosion problem. As a result, applying formal methods to large-scale programs is difficult, even though many reduction and abstraction techniques are investigated. Besides, some other problems still cannot be ignored. Model checking is expensive for concurrency bug detection. Symbolic execution may cause false negatives if the exploration of paths is not sufficient. Nevertheless, if the property that should be checked can be modeled in the program, symbolic execution would not cause false negatives.

4 Automated concurrency bug avoidance

Concurrency program bugs are extremely difficult to be detected and fixed. Errors are often hidden in rare memory access interleavings, so most of time we might only need to adopt reasonable solutions to avoid concurrency bugs so that concurrency programs can run correctly without being affected by hidden bugs. The methods of avoiding concurrency bugs are divided into three main categories and deadlock avoidance is usually discussed separately. Rollback-replay is a widely used strategy used by different avoidance methods to avoid and recover from concurrency bugs. As taking rollback-replay strategy, different avoidance methods just differ from each other in the ways of how to go back to a re-execution site and how to by-pass the failure during re-execution. Figure 9 illustrates the generalized framework of rollback-replay approaches.

4.1 Adding synchronization

Adding synchronization is one of the common means to avoid concurrency bugs. By adding synchronization, a code section will be executed with a certain order relationship as expected and an atomic region will be executed atomically to avoid potential violations.

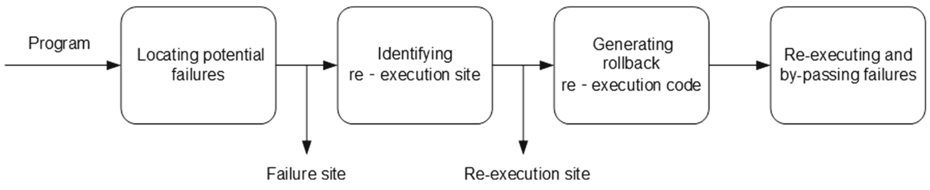

Program
Locating potential failures
Identifying re - execution site
Generating rollback re - execution code
Re-executing and by-passing failures
Failure site
Re-execution site

Fig. 9 The generalized framework of rollback-replay approaches

Autolocker (Mccloskey et al. 2006) conducts lock insertion operations to get pessimistic atomic sections, which refers to using locks to do concurrent control. The pessimistic atomic sections determinately execute atomically without causing deadlocks and data races. However, Autolocker requires programmers to write annotations to make a connection between locks and protected data. One of the closest algorithms to Autolocker is the data-centric approach presented by Vaziri et al. (2006) because they both need manual assistance. It is able to avoid not only traditional but also several high-level data races. This approach statically decides the right location for inserting synchronization to avoid data races. Similarly, AtomRace (Letko et al. 2008) can also heal buggy programs by adding synchronization. AtomRace is a simple and fully automated tool. It is used by an architecture for detecting or even fixing atomicity violations and data races in run-time. After detecting the hidden data races, AtomRace adds synchronization to decrease the probability of data races manifestation.

4.2 Forcing fixed execution order

This kind of method controls thread scheduling and forces programs execute following a fixed order. They directly or indirectly influence thread scheduling and sometimes cause much overhead.

Edelstein et al. (2001) describe a detecting tool and then implement a set of techniques and integrate them into ConTest (Edelstein et al. 2003). ConTest can decrease the probability of data races manifestation. The key ideas of ConTest are to influence the scheduler and to add synchronization actions to prevent data races. AtomRace (Letko et al. 2008) is implemented based on the infrastructure of ConTest. For the healing part of AtomRace, they try to add synchronization as well as influence the scheduler to control threads execution. Although this does not mean to fix bugs, this can guarantee potential bugs only occurs with a lower probability. But AtomRace is different from ConTest on noise injection for detecting concurrency bugs. Their future work focuses on reducing the overhead introduced by ConTest infrastructure and proposing new noise injection heuristics.

Although ConTest and AtomRace have made great progress on concurrency bug avoidance, they still suffer from different problems, such as the limitation in working with polymorphic input behavior and limitation in keeping the system reliable and available after a fault occurs. To address these problems, ASSURE (Sidiroglou et al. 2009) is presented by introducing rescue points, which are positions in existing code generated by fuzzing for handling failures. The rescue points can recover a program from unknown faults. Similarly, Frost (Veeraraghavan et al. 2011) can also survive the first occurrence of unknown bug. Frost detects data races and controls thread scheduling to protect a program from data races with complementary thread schedules.

In general, both ASSURE and Frost require checkpointing the whole program states and once a failure occurs, all threads have to roll back. In order to avoid such sophisticated changes to operating systems, Zhang et al. present ConAir. ConAir (Zhang et al. 2013) can be used to help concurrent programs survive from bugs and fix bugs with temporary patches as well. ConAir consists of three key parts: (1) statically identifying potential erroneous code regions, (2) statically identifying the idempotent code regions around each bug, and (3) inserting rollback code for recovery around the idempotent code regions. ConAir is able to handle a large number of concurrency bugs.

To solve the limitations of software-only approaches, two techniques are proposed with hardware support. Yu et al. (2009) describe a design for an interleaving constrained shared-memory multi-processor. This design aims to eliminate untested interleavings in order to avoid concurrency bugs. They consider that untested interleavings may cause concurrency bugs with a high probability, so they use Predecessor Set (PSet) to encode the set of tested correct interleavings in a program's binary executable. The evaluation with several real-world programs shows that the system is able to avoid data races, atomicity violations, and even some other concurrency bugs with different structures. Atom-Aid (Lucia et al. 2008) also needs help of hardware. It avoids atomicity violations by putting all memory operations of an atomicity region inside the same chunk. It can smartly decide the right location to insert a chunk boundary before the potential atomicity violation happens.

Furthermore, concurrency bug avoidance has special significance to the deployed systems, which have a strict requirement against performance. However, those hardware-dependent approaches may cause complex and expensive changes to hardware so that they will induce excessive overhead. Aviso (Lucia and Ceze 2013) can overcome these challenges and avoid concurrency bugs caused by buggy execution scheduling. The system first monitors the execution of a given program; once a failure happens, it will record a history of events and control the scheduling to avoid the same failure circumstance later in the execution.

4.3 Software transaction memory

Software transaction memory (STM) treats shared memory accesses as transactions in order to execute a set of operations in shared memory as a whole. When a concurrency bug is detected, STM will roll back a transaction and discard memory updates to avoid bugs.

Transactional memory was first introduced by Herlihy et al. (1993). They want to make lock-free synchronization as efficient as mutual exclusion. Their idea has many attractions, but it would be hard to use the transactional memory directly because it needs identification of transactional objects using a type system. As a result, Harris and Fraser (2003) present a technique cooperating with the oldest proposals for concurrency control, Hoare's conditional critical regions (CCRs), to avoid concurrency bugs. They map CCRs to a STM for allowing concurrency programs running without deadlocks. However, the previous two studies both lack a function to judge whether a transaction needs to block or not. Harris et al. (2005) then present a function called *retry* to solve this problem and propose a concurrency model that addresses the problem of composing concurrency abstractions together. *Retry* is a new modular form of blocking function and composes any blocking transactions sequentially. Herlihy et al. (2006) propose DSTM2, which is a Java software library, and is also a framework to implement object-based STM. DSTM2 provides a new kind of thread to execute transactions and define methods, which can create new atomic classes. Compared to previous STM implementations, which are implemented via changing compilers and/or run-time systems, implementations based on DSTM2 are more flexible and easier to distribute.

Ananian et al. (2006) propose unbounded transactional memory and the implementation UTM based on the hardware transactional memory (HTM). They address the constraints in HTM that are about transaction size restriction and also guarantee the efficiency of running small transactions. Both HTM and UTM can be used to ensure atomicity in concurrency programs and avoid bugs caused by locks. However, UTM adds a pointer to every memory block and a linked-list log of reads and writes for its implementation. Consequently, UTM's implementation is too complex. Moore et al. then propose LogTM (Moore et al. 2006) to address the limitations of UTM by developing a new and faster TM system. LogTM also improves the weakness of HTM in transactions size so that it can be applicable to large transactions.

Grace (Berger et al. 2009) is a system to eliminate concurrency bugs at run time and its key idea is to turn threads into processes. It implements STM and treats threads as processes. Grace can make a concurrent program execute determinately and eliminate different concurrency bugs through changing lock operation, committing state changes, and controlling threads execution. With the trend that multi-variable concurrency bugs become more common than expected, there is a need for a general solution to avoid both single- and multi-variable concurrency bugs. As a more general architecture that supports bug detection and avoidance together, ColorSafe (Lucia et al. 2010) dynamically detects and avoids atomicity violations. It compares variables to colors and groups variables into colors. To detect and avoid atomicity violations, ColorSafe monitors suspicious threads executions which may cause atomicity violations and inserts ephemeral transactions to guard against bugs.

4.4 Deadlock avoidance and recovery

Many researchers address deadlocks independent of other concurrency bugs because deadlocks are caused by improper high-level synchronization while other concurrency bugs are mainly caused by incorrect memory access orders. Therefore, we summarize the avoidance strategy for deadlocks separately in this subsection and classify all of these approaches into four types according to their different fixing mechanisms.

4.4.1 Type and effect system methods

Type and effect system statically enforce all threads to obey a global lock-acquisition ordering for deadlocks avoidance. Boudol (2009) present a type and effect system that is used to create semantics to avoid deadlocks for a concurrent programming language. They define deadlock-free semantics. This may not avoid deadlocks when non lexically scoped locking operations are supported. To address the problems, Gerakios et al. (2011) propose a type and effect system that can dynamically avoid deadlocks. The method first collects information of the order of lock and unlock operations, then statically analyzes and computes the locking order. The information can guide the method to avoid deadlocks. Gordon et al. (2012) present a technique called Lock capabilities. Unlike Gerakios et al.'s approach, lock capabilities do not enforce a total order. It can be embedded in a type system used as locking protocol for checking if a concurrent program is deadlock free. One feature of Lock capabilities is that they allow concurrent programs to acquire locks with flexible orders so that they are well suited for fine-grained locking.

4.4.2 Petri nets methods

Petri nets and control theory are also employed to analyze the whole program and some control logic is inserted into programs to avoid deadlock. A project called Gadara Nets (Wang et al. 2009) is a class of Petri nets and utilizes Petri nets to model concurrent programs. This method obtains information when programs are compiled to build Petri net model. Then, Liao and Wang (2013) propose an optimization of ordinary Gadara Nets in eliminating deadlocks with minimally restrictive control logic. They use Petri nets to model multithreaded programs. This method improves the correctness and guarantees the efficiency and scalability compared to previous work.

4.4.3 Run-time methods

The last type of method usually detects and avoids deadlocks at run time. This type of method is presented based on the observation that many bugs have a relation to the execution environment so that run-time avoidance and recovery are more efficient. Qin et al. (2005) present a technique called Rx for safe and transparent software bug detection. This technique can deal with a wider range of bugs than Gadara and its core idea is inspired by allergy treatment in real life. If a deadlock is detected, the technique rolls back to a recent checkpoint and executes the erroneous part again but with thread controlling based on the failure pattern. The whole detection and recovery process are transparent, so Rx is safe enough. Although Rx performs effectively and efficiently in recovering from software failures, it still suffers from some weakness like false positives. One way to deal with this is to save deadlock information and leverage this information to avoid those deadlocks during their second occurrence. Therefore, Jula et al. propose an approach (Jula and Candea 2008; Jula et al. 2008) based on deadlock immunity. Deadlock immunity describes a program's ability that can protect remaining code from deadlocks that have manifested and been caught in the past observed executions. This approach can detect deadlocks when the tested program is running and once a deadlock is detected, the approach will save the pattern of the deadlock so that the approach controls thread scheduling to avoid the deadlock.

Grace (Berger et al. 2009) is also a run-time system. It forces concurrent threads to execute determinately and sequentially based on a sequential commit protocol with virtual memory protection. Moreover, it lets programs be deadlock-free by converting lock operations to no-ops. Sammati (Pyla and Varadarajan 2010), another run-time system with no false positives or negatives, is a language-independent run-time system that automatically detects and avoid deadlocks in concurrent programs. The whole process does not need any manual operations or modification to the source code. The key point of Sammati is that when every lock acquisition operation happens, Sammati leverages a deadlock detection algorithm to check if there is a deadlock. If a deadlock is detected, the algorithm will roll back to the lock acquisition, causing the deadlock and yield memory updates.

4.5 Conclusions of concurrency bug avoidance methods

In summary, concurrency bug avoidance faces many challenges and therefore a number of methods have been proposed. Adding synchronization can avoid concurrency bugs during compilation and bring a small negative impact on the original programs, but they rely a lot on manual assistance or other detectors' reports. Forcing a fixed thread execution order directly or indirectly influences thread scheduling, causing much overhead. When they indirectly influence thread scheduling, they just decrease the probability and danger degree of concurrency bugs, which means that they cannot totally avoid bugs to some extent. STM methods with low overhead would not be developed widely as they cannot handle concurrency bugs with I/O operations.

For all of these four types of deadlock avoidance and recovery methods, each has their own advantages and characteristics. Type-system methods cause less overhead but need manual involvement to carry out the techniques. Effect system methods have a wider applicable range. Petri nets methods have high efficiency and correctness but bad scalability. Instead, run-time systems are more flexible but they cause much overhead.

5 Automated concurrency bug fixing

Researchers have put a lot of effort into concurrency bug detection, exposing, and avoidance. However, without fixing, it does not mean a real improvement of concurrent programs. Consequently, concurrency-bug fixing is critical and challenging. Fixing operations may also introduce new bugs with a high probability. A survey from Microsoft (Godefroid and Nagappan 2008) shows that over 60% of the respondent have to deal with concurrency issues on a monthly basis and most concurrency bugs are hard to fix. It often takes days of work to analyze and fix a single concurrency bug. To fix concurrency bugs, developers are faced with several challenges, such as determining the root causes of bugs, enforcing a specific synchronization felicitously, and so on. The principles for ensuring the correctness of fixing concurrency bugs are no introduction of new bugs, no significant degradation of performance, and no severe sacrifice of code readability.

5.1 The framework of automated concurrency bug fixing

Many solutions for automatically fixing concurrency bugs have been proposed and evaluated by researchers. The whole process of automated concurrency bug fixing usually involves the following four steps. Firstly, the concurrency fault must be located. After reporting a concurrent bug, a fix strategy is proposed and patches for every single bug are generated. Second, patches for a set of related bugs need to be merged and optimized for better performance and readability. Third, each patch must be tested to make sure that it can fix the buggy program correctly without introducing any new bugs. Last, good-quality patches can be generated automatically. Figure 10 illustrates the framework of fixing approaches.

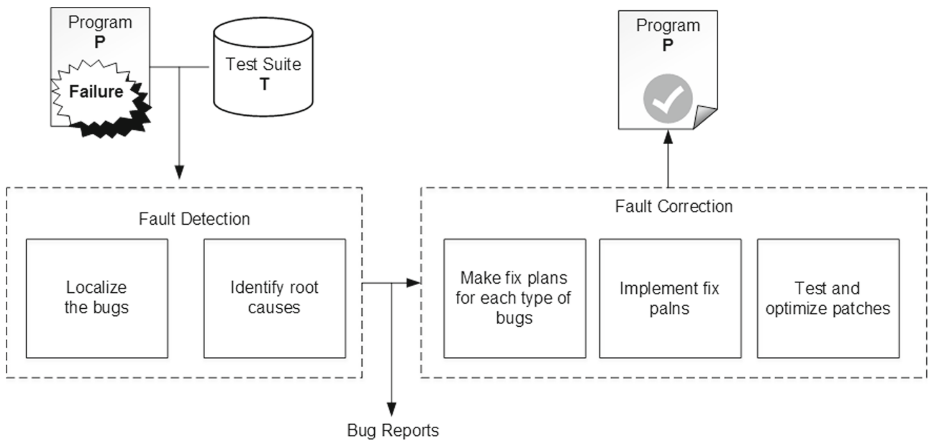


Fig. 10 The framework of fixing approaches

5.2 Concurrency bug localization

The preliminary step of fixing concurrency bugs is to locate them. Some of the techniques detect bugs by effective strategies, yet there are still some other techniques working with existing effective concurrency-bug detectors to assist them. The concurrency bug detectors provide bug reports including interleavings, which induce program failures, to fix techniques as a basis of fixing process.

Prvulovic and Torrellas show that the rollback capabilities of Thread-Level Speculation (TLS) can be extended to support rolling back buggy executions and characterizing bugs. They introduce ReEnact (Prvulovic and Torrellas 2003), a TLS-based debugging framework, which is able to detect, characterize, and repair data races. Its core idea of detecting data races is using TLS to check if two thread slices are unordered. ReEnact makes a try to possibly repair data races, then Krena et al. (2007) propose an algorithm for healing data races on-the-fly. They use Eraser algorithm to firstly detect data races. The main principle of Eraser algorithm is that for each variable the Eraser algorithm maintains its state and the set of candidate locks. When there is a shared variable and the set of candidates locks becomes empty, the algorithm can detect a race. Similar to Křena et al.'s work, some subsequent approaches also find bug root causes with some other tools. AFix (Jin et al. 2011) uses CTrigger (Park et al. 2009; Lu et al. 2011b) bug reports to discover atomicity variable. AFix even modifies CTrigger to output all possible combinations of atomicity-violation triples and the complete call stack for each atomicity-violation related statements. On the basis of AFix, CFix (Jin et al. 2012) aims at automatically fixing different types of concurrency bugs. Similar to AFix, it continues the fixing process with the help of bug reports that just need to report the execution order of bug-related code.

AFix and CFix both attempt to fix existing concurrency bugs, nevertheless, Weeratunge et al. (2011) propose a technique that can fight against unknown concurrency bugs. The technique first analyzes a set of traces and for each pair of consecutive memory accesses of the local thread, they check if accesses from a remote thread interleave the pair of accesses. If it does, an atomicity violation is detected. The above techniques all operate on bytecode or the programs' original source to detect bugs. Different from these techniques, Raft (Smith 2013) performs on programs' machine code with minimal higher-level knowledge of programs' structure or environment. Raft consists of a list of techniques for discovering, characterizing, reproducing, and fixing concurrency bugs. Raft verifies potential bugs by representing all instructions involved in a crash as a dynamic Control Flow Graph (CFG) and get approximate instructions from the CFG by reverse engineering. After analyzing the difference between approximates and crashes, a verification condition will be achieved. For more general applications with little manual effort, ConcBugAssist (Khoshnood et al. 2015) is presented as a diagnosing and repairing tool for concurrent programs. Giving a buggy concurrent program, ConcBugAssist uses a symbolic analysis method to compute a set of minimal inter-thread ordering constraints that cause violations. The set may be treated as the precondition of bug fixing.

5.3 Fix-strategy planning and implementation

The next step is fix-strategy planning and implementation. Different kinds of concurrency bugs need different strategies to fix them. Fix strategies include many manners aiming at eliminating erroneous interleaving patterns to make concurrent programs be fixed correctly.

Based on the fix strategies, fix techniques then decide how to implement the strategies and enforce synchronization properly.

Isolation in concurrent programs can guarantee that unexpected interleavings between threads will not occur and induce program failures. For a concurrent program, ISOLATOR (Rajamani et al. 2009) helps to detect and avoid isolation violations with a locking discipline and a set of threads that obey the locking discipline. It associates every lock and exploits page protection to guarantee isolation. Based on the experience with Gadara (Wang et al. 2009), Kelly et al. find that discrete control theory is helpful in solving concurrency problems. They (Kelly 2009) use control engineering to address the challenges of fixing concurrency bugs. Inspired by the great use of genetic programming in sequential software debugging, Bradbury and Jalbert (2010) use genetic programming to fix concurrency bugs in an incremental manner. They mutate the program at every generation and use a fitness function to compare the mutation with the parent program and choose part of mutations into the next generation.

As a totally automated fixing tool, AFix (Jin et al. 2011) focuses on fixing single-variable atomicity violations. With the help of bug reports from CTrigger, AFix firstly defines a triplet (p, c, r), where p (preceding), and c (current) from local threads are interleaved by r (remote) from remote threads. AFix further identifies atomicity violations by a set of (p, c, r) triples. Then, AFix puts p and c into a critical region and put r into another one so that the two regions are protected by a lock respectively which is mutually exclusive with each other. With the information of failure-inducing interleaving, for different types of concurrency bugs such as atomicity violations, order violations, data races, and abnormal def-use problems, CFix (Jin et al. 2012) enforce mutual exclusion for mutual-exclusion problems and all A-B and first A-B order relationships for order problems. Liu et al. evaluate AFix on large real systems and declare that AFix may incur degraded performance and deadlocks. Then they present Axis (Liu and Zhang 2012) to fix atomicity violations by first making a Petri net model for a buggy program, then encoding the Petri net and erroneous statements as a set of control constraints. Finally, an augmented Petri net is generated that matches the patched code that we want.

To reduce the nondeterminacy of concurrent programs, Huang and Zhang (2012) privatize the shared data races using a path and context-sensitive execution privatization technique to alleviate concurrency problems. Previous techniques are context oblivious focusing only on the buggy statements and ignoring the context. It is nontrivial for Grail (Liu et al. 2014a) to realize such context-aware bug fixing. Grail fixes concurrency bugs with both correctness and optimality guarantees because it simultaneously synthesizes and optimizes lock-based synchronization. Grail takes both the incorrect statements and context into consideration. Joshi and Lal (2014) infer atomic code regions for fixing bugs. They assume that if a program's threads are not interleaved by any thread, the program is correct. Their inferring process is based on this assumption.

Some techniques conduct different studies for better fixing results. Yin et al. (2011) conduct a characteristic study on incorrect bug fixes from real-world large programs. They find that there are at least 14.8– 24.4% incorrect fixes which programmers and reviewers usually have no good solutions to deal with. After making an empirical study of real-world concurrency bugs, Cĕrný et al. discover that most previous fixing techniques differ a lot from the strategies used by programmers. To be more practical, Černý et al. (2013) use program synthesis to explore different kinds of program transformations without sacrificing program semantics and remove bugs taken partial-order traces as counterexamples. Furthermore, they present another synthesis algorithm (Černý et al. 2014) to avoid introducing new concurrency bugs by repair algorithms. According to a set of good traces and bad traces of program execution, the algorithm learns the constraints which can be used to eliminate the bad traces. In order to generate patches with high quality, Liu et al. (2016) first study 77 manual patches for real-world concurrency bugs and design fix strategies guided by the study results. After having a better understanding of patching concurrency bugs, they propose HFix that adds thread-join operations to enforce order relationship and leverage existing synchronization operations to fix order problems and atomicity violations. These two strategies generate better and simpler patches than prior techniques.

5.4 Patch verification and optimization

The third step is patch verification and optimization. Fix techniques first generate patches for every single bug. Then, a lot of candidate patches need to be tested and selected for their performance in correctness, overhead costing, and readability. Besides, a set of patches of related bugs can also be merged to improve the simplicity of patches and decrease the probability of introducing deadlocks.

Bug reports usually include multiple bugs that cause program failures. For all bugs in reports, AFix (Jin et al. 2011) designs a patch for each of them. Thus, AFix tries to remove redundant patches and merge related patches to improve code readability, program performance, and correctness, as well as reducing deadlock risk. In addition, all patches generated by AFix need to be verified twice. The first one is verified by CTrigger to check whether the bug has been fixed. The second verification is implemented by AFix for checking if a patch is able to fix bugs. CFix (Jin et al. 2012) tests patches by first checking the correctness through static analysis and interleaving testing, then comparing the performance for all patches passing correctness testing, finally observing timeout issue for deadlock detection.

For a wider variety of bugs, Kelk et al. (2013) propose ARC, which can repair deadlocks and data races automatically for concurrent Java programs. Because there may be some unnecessary synchronization inserted by accident, ARC conducts an optimization after a fix operation to improve the performance of the program. They use a fitness function to compute the degree of improvement with respect to program running time and number of context switches. After a core part of the algorithm, Flint (Liu et al. 2014b) performs optimizations for performance with no harm to correctness. Optimizations hold the key idea of fast-path to promote the minimal implementation of Flint, which consists of only one Map operation. After optimizing, the code will be more efficient than before. HFix (Liu et al. 2016) first analyzes a lot of manual patches for real-world concurrency bugs in detail. Based on the analysis, it automatically fixes concurrency bugs guided by empirical study findings. Finally, HFix merges at least two patches, which are applied the same fix strategy.

5.5 Automated deadlock fixing

A program will be blocked when a deadlock occurs. Among all the methods we have introduced above to automatically fix concurrency bugs, the majority of them need to insert new locks into original programs for concurrency-bug fixing. Inserting new locks can fix some non-deadlock concurrency bugs by serializing those threads that are related to these bugs. On the other hand, it may also introduce new deadlock instead of fixing deadlock bugs. As a result, a few techniques, which only focus on deadlock fixing, are proposed and evaluated well on real-world programs.

Jula et al. (2008) present a technique by which a concurrent program can be defended against deadlocks after they are detected at the first time. When a deadlock appears for the first time, the technique records the erroneous interleaving patterns, then it avoids the manifestation of the buggy patterns by inserting instrumentation codes. Nirbuchbinder et al. (2008) aim to protect concurrency programs from deadlocks. The key idea is to force a sequential execution of the code regions involved in deadlocks. After summarizing the limitations of previous techniques on handling deadlocks which can not scale to large applications and have high overhead, Cai et al. (2013) plan to implement an active lock assignment, which refers to making a thread acquire the corresponding lock when they are involved in a deadlock. This strategy enables a thread involved in a deadlock to pre-acquire a corresponding lock, which can avoid a circular waiting blocking the program. In the follow-on work, Cai and Cao (2016) present an approach DFixer to pre-acquire a lock so as to break one necessary condition of a deadlock. The experimental results show that DFixer can successfully fix all 20 deadlocks.

5.6 Conclusions of concurrency bug-fixing methods

Concurrency bug fixing becomes capable for all common types of concurrency bugs. Analyzing the root cause of concurrency bugs is the first important step of fixing them. The analyzing results' accuracy and correctness may directly influence the fixing results. Fix strategies should be directed against bug types and the implementation should decide where and how to enforce the strategies so as to generate patches. Patch testing and optimizing aims at getting the best patch considering program correctness, performance, and patch simplicity. All steps link with each other and fuse into the integrity throughout the whole process of automated concurrency bug fixing.

6 Benchmark summary

Most automated concurrency bug exposing, detection, avoidance, and fixing methods use real-world programs for empirical study and evaluation. After analyzing the collected papers, we conclude the benchmarks that are the most actively used by researchers (at least eight times) as shown in Table 1. Table 1 lists the program name, programming language, line of code, cumulative usage count, and related literature for each of the benchmarks. The benchmarks come from different areas, ranging from data-centric applications, long-running applications, computation-centric applications, to web-centric applications.

7 The active research groups and their contribution

This section summarizes the active research groups in the field of concurrency bug exposing, detection, avoidance, and fixing in alphabetic order of their names.

7.1 Cai et al.

Cai et al. made many contributions on deadlocks detection, testing, and fixing. They concentrate on the research on deadlocks and present a lot of excellent research achievement.

Table 1 Benchmarks used by different methods (LOC means lines of code)

Program	Language	LOC	Usage count	Related literatures
MySQL	C	1900k	30	Cai and Cao (2016), Cai and Chan (2012), Cai and Lu (2016), Cai et al. (2014), Cai and Chan (2014), Chew (2009), Jin et al. (2011), Jin et al. (2012), Khoshnood et al. (2015), Liu et al. (2016), Lu et al. (2007), Lu et al. (2007), Lu et al. (2011a), Lu et al. (2011b), Lucia et al. (2008), Lucia et al. (2010), Lucia et al. (2011), Park et al. (2009), Park et al. (2012), Qin et al. (2005), Shi et al. (2010), Veeraraghavan et al. (2011), Weeratunge et al. (2011), Xu et al. (2005) and Yu and Narayanasamy (2009), Yu et al. (2012), Zhang et al. (2016), Zhang et al. (2013), Zhang et al. (2011), Zhang et al. (2010)
Apache	C	345k	22	Chew (2009), Jin et al. (2011), Jin et al. (2010), Jin et al. (2012), Kasikci et al. (2013), Khoshnood et al. (2015), Liu et al. (2016), Lu et al. (2011b), Lu et al. (2007), Lucia et al. (2010), Lucia et al. (2008), Lucia et al. (2011), Park et al. (2009), Qin et al. (2005), Shi et al. (2010), Veeraraghavan et al. (2011), Weeratunge et al. (2011), Xu et al. (2005), Yu and Narayanasamy (2009) and Yu et al. (2012), Zhang et al. (2016), Zhang et al. (2010)
Mozilla	C	3400k	18	Burckhardt et al. (2010), Jin et al. (2011), Jin et al. (2010), Jin et al. (2012), Khoshnood et al. (2015), Liu et al. (2016), Lu et al. (2007), Lu et al. (2007), Lu et al. (2011a), Lucia et al. (2010), Lucia et al. (2011), Park et al. (2009) and Shi et al. (2010), Weeratunge et al. (2011), Yu and Narayanasamy (2009), Zhang et al. (2013), Zhang et al. (2011), Zhang et al. (2010)
Pbzip2	C++	2k	18	Burckhardt et al. (2010), Jin et al. (2011), Jin et al. (2010), Jin et al. (2012), Kasikci et al. (2013), Kasikci et al. (2015), Liu et al. (2016), Lu et al. (2011b), Lucia et al. (2011), Park et al. (2009), Park et al. (2012), Shi et al. (2010), Veeraraghavan et al. (2011), Weeratunge et al. (2011), Yu and Narayanasamy (2009), Yu et al. (2012), Zhang et al. (2016), Zhang et al. (2010)
FFT	C	1.2k	10	Jin et al. (2011), Jin et al. (2010), Jin et al. (2012), Liu et al. (2016), Lu et al. (2011b), Park et al. (2009), Zhang et al. (2010), Zhang et al. (2016), Zhang et al. (2013), Zhang et al. (2011)
SQLite	C	67k	8	Cai and Cao (2016), Cai and Chan (2012), Cai and Chan (2014), Cai and Lu (2016), Cai et al. (2014), Kasikci et al. (2013), Kasikci et al. (2015), Zhang et al. (2013)
Transmission	C	95k	8	Jin et al. (2012), Khoshnood et al. (2015), Liu et al. (2016), Shi et al. (2010), Zhang et al. (2016), Zhang et al. (2013), Zhang et al. (2011), Zhang et al. (2010)
Aget	C	1.2k	8	Kasikci et al. (2013), Lucia et al. (2010), Lucia et al. (2011), Park et al. (2012), Weeratunge et al. (2011), Yu and Narayanasamy (2009), Yu et al. (2012), Zhang et al. (2011)

They proposed MagicFuzzer (Cai and Chan 2012), Magiclock (Cai and Chan 2014), and other deadlock detectors especially for large-scale concurrent programs. Their latest work, DFixer (Cai and Cao 2016), shows very prominent performance on deadlock fixing.

7.2 Flanagan et al.

These authors studied a lot on data race and atomicity violation detection. They also gave a series of construction methods. They presented a type system (Flanagan and Freund 2000) for catching race conditions statically and described *rccjava*, an implementation of this system for Java. They later presented a sound and complete atomicity checker that finds and checks atomicity violations found by other tools.

7.3 Huang, Liu, and Zhang et al.

This research group has worked in avoiding and fixing different kinds of concurrency bugs. They presented Axis (Liu and Zhang 2012), Flint (Liu et al. 2014b), and Grail (Liu et al. 2014a), a set of useful algorithms and tools for automatically fixing concurrency bugs. Flint is the first general algorithm for automatically transforming nonlinearizable compositions of Map operations into atomic compositions.

7.4 Lu et al.

Lu et al. work in many fields of dealing with concurrency bugs, including concurrency bug exposing, detection, recovery, prevention, and fixing. They used data-mining techniques to detect and fix concurrency bugs (Lu et al. 2007; Liu et al. 2016). Aiming at different types of concurrency bugs, they presented a set of automated fixing tools, such as AFix (Jin et al. 2011), CFix (Jin et al. 2012), and HFix (Liu et al. 2016). Their study on real-world concurrency bug characteristics (Lu et al. 2008) provides a comprehensive study of real-world concurrency bugs, examining their pattern, manifestation, fix strategy, and other characteristics. A lot of research benefits from their work in various aspects.

7.5 Lucia et al.

These authors made an outstanding contribution to concurrency bug avoidance. They proposed Atom-Aid (Lucia et al. 2008). It is the first approach using chunk boundaries to avoid atomicity violations. They also proposed ColorSafe (Lucia et al. 2011), an architecture that provides dynamic bug avoidance for a wide variety of bugs, and Aviso (Lucia and Ceze 2013), a software-only system that avoids concurrency failures by empirically determining fault-free execution schedules.

7.6 Park et al.

Park et al. contributed to the fault localization of concurrent programs. They proactively located concurrency bugs through shared memory access patterns and they presented Falcon, the first technique to both report and rank patterns. They later presented UNICORN, the first unified technique that detects and ranks non-deadlock concurrency bugs using patterns. UNICORN is based on Falcon and extends detection ability from single-variable concurrency bugs to both single- and multi-variable concurrency bugs.

8 Conclusions and future work

Over the last couple of decades, concurrency bugs have been widely studied and a lot of significant research efforts have been devoted. An increasing number of researchers have paid their attention to concurrency bugs, such as concurrency-bug exposing, detection, avoidance, and fixing.

In this article, we make three major contributions: (1) After performing a literature review of more than 100 papers, we make a complete and systematic survey for dealing with concurrency bugs, from concurrency bug exposing, detection, avoidance, to fixing; (2) We summarize the whole process of dealing with concurrency bugs into four main processes according to the difficulty of the procedures. For each process, we survey the key issues, challenges, and the state of art; (3) We summarized the classical benchmarks and active research groups in this active research topic.

Based on our study, we point out a number of research directions on concurrency bug problems in the future:

(1) Deterministic execution is critical to concurrent programs. The non-determinacy of concurrent programs increase the difficulty of reproducing bugs. When a bug is exposed in an execution, we hope it can be exposed with the same input the next time. This will make it much more efficient to detect and expose concurrency bugs.
(2) More efficient techniques for recording and replaying concurrent programs need to be developed. The execution with specific interleaving scheme of a concurrent program is the key for dealing with concurrency bugs from detection to fixing. We need to record the executions and replay any one of them for study.
(3) Due to the huge state space for the thread interleavings, there is a strong need for more effective detection and fixing techniques. Firstly, some detection and fixing techniques for traditional bugs could be cooperated to deal with concurrency bugs. On the other hand, we will explore some new methods based on the features of concurrency bugs.
(4) More empirical studies on real concurrency bugs should be conducted and these bugs should be employed to enrich the benchmark suite. Before studying real bugs, more tools should be designed to automatically extract bugs from concurrency bug databases. This would facilitate programmers to understand the root cause of concurrency bugs after mining bug repositories and finding some patterns from these bugs.
(5) Another fruitful research area in concurrency bugs is cooperating software and hardware. Software-only systems are applicable for different platforms, but may have a negative influence on program performance. Some systems modify hardware to improve efficiency with bad applicability and generality. We can propose some methods to combine hardware and software to take full advantage of them.

Acknowledgements This work is partly supported by projects from National Natural Science Foundation of China, with the project number '61202030', '71502125', '61373012', and '61202006'. The authors also thank anonymous reviewers for their constructive comments.

References

Abadi, M., Flanagan, C., & Freund, S.N. (2006). Types for safe locking: static race detection for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *28*, 207–255.

Ananian, C.S., Asanovic, K., Kuszmaul, B.C., Leiserson, C.E., & Lie, S. (2006). Unbounded transactional memory. In *International conference on high-performance computer architecture* (pp. 59–69).

Berger, E.D., Yang, T., Liu, T., & Novark, G. (2009). Grace: Safe multithreaded programming for C/C++. In *ACM Sigplan Notices*, (Vol. 44 pp. 81–96): ACM.

Boigelot, B., & Godefroid, P. (1996). Model checking in practice: an analysis of the access.bus protocol using spin. In *Proceedings of the 3rd international symposium of formal methods Europe on industrial benefit and advances in formal methods* (pp. 465–478).

Boudol, G. (2009). A deadlock-free semantics for shared memory concurrency. In *International colloquium on theoretical aspects of computing* (pp. 140–154).

Bradbury, J.S., & Jalbert, K. (2010). Automatic repair of concurrency bugs. In *International symposium on search based software engineering* (pp. 1–2).

Bron, A., Farchi, E., Magid, Y., Nir, Y., & Ur, S. (2005). Applications of synchronization coverage. In *Proceedings of the 10th ACM SIGPLAN symposium on principles and practice of parallel programming* pp. (206–212). ACM.

Burckhardt, S., Kothari, P., Musuvathi, M., & Nagarakatte, S. (2010). A randomized scheduler with probabilistic guarantees of finding bugs. *ACM Sigplan Notices*, *45*, 167–178.

Cadar, C., Dunbar, D., & Engler, D. (2008). Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Usenix symposium on operating systems design and implementation, OSDI 2008* (pp. 209–224). California, USA: Proceedings.

Cai, Y., & Cao, L. (2016). Fixing deadlocks via lock pre-acquisitions. In *Proceedings of the 38th international conference on software engineering* (pp. 1109–1120). ACM.

Cai, Y., & Chan, W.K. (2012). Magicfuzzer: scalable deadlock detection for large-scale applications. In *International conference on software engineering* (pp. 606–616).

Cai, Y., & Chan, W.K. (2014). Magiclock: scalable detection of potential deadlocks in large-scale multithreaded programs. *IEEE Transactions on Software Engineering*, *40*, 266–281.

Cai, Y., & Lu, Q. (2016). Dynamic testing for deadlocks via constraints. *IEEE Transactions on Software Engineering*, *42*(9), 825–842.

Cai, Y., Chan, W.K., & Yu, Y.T. (2013). Taming deadlocks in multithreaded programs. In *International conference on quality software* (pp. 276–279).

Cai, Y., Wu, S., & Chan, W. (2014). Conlock: a constraint-based approach to dynamic checking on deadlocks in multithreaded programs. In *Proceedings of the 36th international conference on software engineering* (pp. 491–502). ACM.

Černỳ, P., Henzinger, T.A., Radhakrishna, A., Ryzhyk, L., & Tarrach, T. (2013). *Efficient synthesis for concurrency by semantics-preserving transformations*. Lecture Notes in Computer Science (pp. 951–967).

Černỳ, P., Henzinger, T.A., Radhakrishna, A., Ryzhyk, L., & Tarrach, T. (2014). Regression-free synthesis for concurrency. In *International conference on computer aided verification* (pp. 568–584). Springer.

Chew, L. (2009). *A system for detecting, preventing and exposing atomicity violations in multithreaded programs*. University of Toronto.

Chew, L., & Lie, D. (2010). Kivati: fast detection and prevention of atomicity violations. In *European conference on computer systems, proceedings of the European conference on computer systems, EUROSYS 2010* (pp. 307–320). Paris, France.

Choi, J.D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., & Sridharan, M. (2002). Efficient and precise datarace detection for multithreaded object-oriented programs. *ACM Sigplan Notices*, *37*, 258–269.

Clarke, E.M., & Wing, J.M. (1996). Formal methods: State of the art and future directions. *ACM Computing Surveys*, *28*, 626–643.

Clarke, E.M., Grumberg, O., Hiraishi, H., Jha, S., Long, D.E., McMillan, K.L., & Ness, L.A. (1995). Verification of the futurebus+ cache coherence protocol. *Formal Methods in System Design*, *6*(2), 217–232.

Deng, D., Zhang, W., & Lu, S. (2013). Efficient concurrency-bug detection across inputs. *ACM Sigplan Notices*, *48*, 785–802.

Deng, D.D., Jin, G.L., Marc, D.K., Ang, L.I., Ben, L., Shan, L.U., Shanxiang, Q.I., Ren, J.L., Karthikeyan, S., & Song, L.H. (2015). Fixing, preventing, and recovering from concurrency bugs. *Science China Information Sciences*, *58*, 1–18.

Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G., & Ur, S. (2001). Multithreaded Java program test generation. In *Joint ACM-iscope conference on Java grande* (pp. 111–125).

Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby, G., & Ur, S. (2003). Framework for testing multithreaded Java programs. *Concurrency and Computation: Practice and Experience*, *15*, 485–499.

Engler, D.R., & Ashcraft, K. (2003). Racerx: effective, static detection of race conditions and deadlocks. In *ACM SIGOPS operating systems review* (Vol. 37, 237–252). ACM.

Flanagan, C., & Freund, S.N. (2000). Type-based race detection for Java. *ACM Sigplan Notices*, *35*, 219–232.

Flanagan, C., Freund, S.N., & Yi, J. (2008). Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. *ACM SIGPLAN Notices*, *43*(6), 293–303.

Gerakios, P., Papaspyrou, N., & Sagonas, K. (2011). A type and effect system for deadlock avoidance in low-level languages. In *Proceedings of the 7th ACM SIGPLAN workshop on types in language design and implementation* (pp. 15–28). ACM.

Godefroid, P. (1997). Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on principles of programming languages* (pp. 174–186).

Godefroid, P., Klarlund, N., & Sen, K. (2005). Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation* (pp. 213–223).

Godefroid, P., & Nagappan, N. (2008). Concurrency at Microsoft: An exploratory survey. In *CAV workshop on exploiting concurrency efficiently and correctly*.

Gordon, C.S., Ernst, M.D., & Grossman, D. (2012). Static lock capabilities for deadlock freedom. In *Proceedings of the 8th ACM SIGPLAN workshop on types in language design and implementation* (pp. 67–78). ACM.

Harris, T., & Fraser, K. (2003). Language support for lightweight transactions. *ACM Sigplan Notices*, *38*, 388–402.

Harris, T., Marlow, S., Peyton-Jones, S., & Herlihy, M. (2005). Composable memory transactions. In *Proceedings of the 10th ACM SIGPLAN symposium on principles and practice of parallel programming* (pp. 48–60). ACM.

Herlihy, M., Eliot, J., & Moss, B. (1993). Transactional memory: architectural support for lock-free data structures. In *International symposium on computer architecture* (pp. 289–300).

Herlihy, M., Luchangco, V., & Moir, M. (2006). *A flexible framework for implementing software transactional memory*. ACM Sigplan Notices (p. 41).

Huang, J., & Zhang, C. (2012). Execution privatization for scheduler-oblivious concurrent programs. In *ACM international conference on object oriented programming systems languages and applications* (pp. 737–752).

Jin, G., Thakur, A., Liblit, B., & Lu, S. (2010). Instrumentation and sampling strategies for cooperative concurrency bug isolation. *ACM Sigplan Notices*, *45*, 241–255.

Jin, G., Song, L., Zhang, W., Lu, S., & Liblit, B. (2011). Automated atomicity-violation fixing. *ACM Sigplan Notices*, *46*, 389–400.

Jin, G., Zhang, W., Deng, D., Liblit, B., & Lu, S. (2012). Automated concurrency-bug fixing. In *Usenix conference on operating systems design and implementation* (pp. 221–236).

Joshi, S., & Lal, A. (2014). *Automatically finding atomic regions for fixing bugs in concurrent programs*. Computing Research Repository.

Joshi, P., Park, C.S., Sen, K., & Naik, M. (2009). A randomized dynamic program analysis technique for detecting real deadlocks. *ACM Sigplan Notices*, *44*, 110–120.

Joshi, P., Naik, M., Sen, K., & Gay, D. (2010). An effective dynamic analysis for detecting generalized deadlocks. In *ACM sigsoft international symposium on foundations of software engineering* (pp. 327–336). NM, USA.

Jula, H., & Candea, G. (2008). A scalable, sound, eventually-complete algorithm for deadlock immunity. In *International workshop on runtime verification* (pp. 119–136). Springer.

Jula, H., Tralamazza, D., Zamfir, C., & Candea, G. (2008). Deadlock immunity: enabling systems to defend against deadlocks. In *Proceedings of the 8th USENIX conference on operating systems design and implementation* (pp. 295–308). USENIX Association.

Kahlon, V., Gupta, A., & Sinha, N. (2006). Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions. In *International conference on computer aided verification* (pp. 286–299).

Kasikci, B., Zamfir, C., & Candea, G. (2013). Racemob: crowdsourced data race detection. In *Twenty-Fourth ACM symposium on operating systems principles* (pp. 406–422).

Kasikci, B., Zamfir, C., & Candea, G. (2015). Automated classification of data races under both strong and weak memory models. *ACM Transactions on Programming Languages & Systems*, *37*, 1–44.

Kelk, D., Jalbert, K., & Bradbury, J.S. (2013). Automatically repairing concurrency bugs with arc. In *International conference on multicore software engineering, performance, and tools* (pp. 73–84). Springer.

Kelly, T. (2009). Eliminating concurrency bugs with control engineering. *Computer*, *42*, 52–60.

Khoshnood, S., Kusano, M., & Wang, C. (2015). Concbugassist: constraint solving for diagnosis and repair of concurrency bugs. In *Proceedings of the 2015 international symposium on software testing and analysis* (pp. 165–176). ACM.

Krena, B., Letko, Z., Tzoref, R., Ur, S., & Vojnar, T. (2007). Healing data races on-the-fly. In *Proceedings of the 2007 ACM workshop on parallel and distributed systems: testing and debugging* (pp. 54–64). ACM.

Kundu, S., Ganai, M.K., & Wang, C. (2010). Contessa: concurrency testing augmented with symbolic analysis. In *Computer aided verification, international conference, CAV 2010* (pp. 127–131). Edinburgh, UK: Proceedings.

Lahiri, S.K., Qadeer, S., & Rakamarić, Z. (2009). Static and precise detection of concurrency errors in systems code using SMT solvers. In *Proceedings of the 21st international conference on computer aided verification* (pp. 509–524).

Lal, A., & Reps, T. (2009). Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, *35*, 73–97.

Letko, Z., Vojnar, T., & Křena, B. (2008). Atomrace: data race and atomicity violation detector and healer. In *Proceedings of the 6th workshop on parallel and distributed systems: testing, analysis, and debugging* (pp. 7:1–7:10). ACM.

Liao, H., & Wang, Y. (2013). Eliminating concurrency bugs in multithreaded software: a new approach based on discrete-event control. *IEEE Transactions on Control Systems Technology*, *21*, 2067–2082.

Liu, P., & Zhang, C. (2012). Axis: Automatically fixing atomicity violations through solving control constraints. In *Proceedings of the 34th international conference on software engineering* (pp. 299–309). IEEE Press.

Liu, P., Tripp, O., & Zhang, C. (2014). Grail: context-aware fixing of concurrency bugs. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering* (pp. 318–329). ACM.

Liu, P., Tripp, O., & Zhang, X. (2014). Flint: fixing linearizability violations. *ACM Sigplan Notices*, *49*, 543–560.

Liu, H., Chen, Y., & Lu, S. (2016). Understanding and generating high quality patches for concurrency bugs. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering* (pp. 715–726). ACM.

Lu, S., Park, S., Hu, C., Ma, X., Jiang, W., Li, Z., Popa, R.A., & Zhou, Y. (2007). MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. *ACM Sigops Operating Systems Review*, *41*, 103–116.

Lu, S., Tucek, J., Qin, F., & Zhou, Y. (2007). Avio: detecting atomicity violations via access-interleaving invariants. *IEEE Micro*, *27*, 26–35.

Lu, S., Park, S., Seo, E., & Zhou, Y. (2008). Learning from mistakes: a comprehensive study on real-world concurrency bug characteristics. In *International conference on architectural support for programming languages and operating systems, ASPLOS 2008* (pp. 329–339). WA, USA.

Lu, S., Park, S., & Zhou, Y. (2011). Detecting concurrency bugs from the perspectives of synchronization intentions. *IEEE Transactions on Parallel and Distributed Systems*, *23*, 1060–1072.

Lu, S., Park, S., & Zhou, Y. (2011). Finding atomicity-violation bugs through unserializable interleaving testing. *IEEE Transactions on Software Engineering*, *38*, 844–860.

Lucia, B., & Ceze, L. (2013). Cooperative empirical failure avoidance for multithreaded programs. In *ACM SIGPLAN notices* (Vol. 48, pp. 39–50). ACM.

Lucia, B., Devietti, J., Strauss, K., & Ceze, L. (2008). Atom-aid: detecting and surviving atomicity violations. *IEEE Micro*, *29*, 73–83.

Lucia, B., Ceze, L., & Strauss, K. (2010). Colorsafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. *ACM SIGARCH Computer Architecture News*, *38*, 222–233.

Lucia, B., Wood, B.P., & Ceze, L. (2011). Isolating and understanding concurrency errors using reconstructed execution fragments. *ACM Sigplan Notices*, *46*, 378–388.

Michael, C.H. (1997). *Why engineers should consider formal methods*. Technical report, NASA Langley Technical Report Server.

Mccloskey, B., Zhou, F., Gay, D., & Brewer, E. (2006). Autolocker: synchronization inference for atomic sections. *ACM Sigplan Notices*, *41*, 346–358.

Moore, K., Bobba, J., Moravan, M.J., & Hill, M. (2006). Logtm: log-based transactional memory. *HPCA*, *27*, 254–265.

Musuvathi, M., & Qadeer, S. (2007). Iterative context bounding for systematic testing of multithreaded programs. *ACM Sigplan Notices*, *42*, 446–455.

Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., & Neamtiu, I. (2008). Finding and reproducing Heisenbugs in concurrent programs. In *Usenix symposium on operating systems design and implementation, OSDI 2008* (pp. 267–280). California, USA: Proceedings.

Naik, M., & Aiken, A. (2007). Conditional must not aliasing for static race detection. In *ACM SIGPLAN notices* (Vol. 42, pp. 327–338). ACM.

Naik, M., Aiken, A., & Whaley, J. (2006). Effective static race detection for Java. *ACM Sigplan Notices*, *41*, 308–319.

Nirbuchbinder, Y., Tzoref, R., & Ur, S. (2008). Deadlocks: from exhibiting to healing. *Lecture Notes in Computer Science*, *5289*, 104–118.

Park, S., Lu, S., & Zhou, Y. (2009). Ctrigger: exposing atomicity violation bugs from their hiding places. *ACM Sigplan Notices*, *44*, 25–36.

Park, S., Vuduc, R., & Harrold, M.J. (2012). A unified approach for localizing non-deadlock concurrency bugs. In *2012 IEEE 5th international conference on software testing, verification and validation* (pp. 51–60). IEEE.

Park, S., Vuduc, R.W., & Harrold, M.J. (2010). Falcon: fault localization in concurrent programs. In *Proceedings of the 32nd ACM/IEEE international conference on software engineering* (Vol. 1, pp. 245–254). ACM.

Prvulovic, M., & Torrellas, J. (2003). Reenact: using thread-level speculation mechanisms to debug data races in multithreaded codes. *ACM Sigarch Computer Architecture News*, *31*, 110–121.

Pyla, H.K., & Varadarajan, S. (2010). Avoiding deadlock avoidance. In *International conference on parallel architecture and compilation techniques* (pp. 75–86).

Qadeer, S., & Rehof, J. (2005). Context-bounded model checking of concurrent software. In *Proceedings of the 11th international conference on tools and algorithms for the construction and analysis of systems* (pp. 93–107).

Qin, F., Tucek, J., Sundaresan, J., & Zhou, Y. (2005). Rx: treating bugs as allergies—a safe method to survive software failures. In *ACM sigops operating systems review* (Vol. 39, pp. 235–248). ACM.

Rabinovitz, I., & Grumberg, O. (2005). Bounded model checking of concurrent programs. In *Proceedings of the 17th international conference on computer aided verification* (pp. 82–97).

Rajamani, S., Ramalingam, G., Ranganath, V.P., & Vaswani, K. (2009). Isolator: dynamically ensuring isolation in concurrent programs. In *International conference on architectural support for programming languages and operating systems, ASPLOS 2009* (pp. 181–192). Washington DC, USA.

Ratanaworabhan, P., Burtscher, M., Kirovski, D., Zorn, B., Nagpal, R., & Pattabiraman, K. (2009). Detecting and tolerating asymmetric races. In *ACM sigplan notices* (Vol. 44, pp. 173–184). ACM.

Rungta, N., Mercer, E.G., & Visser, W. (2009). Efficient testing of concurrent programs with abstraction-guided symbolic execution. In *Model checking software, international SPIN workshop* (pp. 1885–1904). Grenoble, France: Proceedings.

Said, M., Wang, C., Yang, Z., & Sakallah, K. (2011). Generating data race witnesses by an SMT-based analysis. In *International Conference on NASA Formal Methods* (pp. 313–327).

Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., & Anderson, T. (1997). Eraser: a dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, *15*, 391–411.

Sen, K. (2008). Race directed random testing of concurrent programs. *ACM Sigplan Notices*, *43*, 11–21.

Sen, K., Marinov, D., & Agha, G. (2005). Cute: a concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on foundations of software engineering* (pp. 263–272).

Shi, Y., Park, S., Yin, Z., Lu, S., Zhou, Y., Chen, W., & Zheng, W. (2010). Do I use the wrong definition?: Defuse: definition-use invariants for detecting concurrency and sequential bugs. *ACM Sigplan Notices*, *45*, 160–174.

Sidiroglou, S., Laadan, O., Perez, C., Viennot, N., Nieh, J., & Keromytis, A.D. (2009). Assure: automatic software self-healing using rescue points. *ACM Sigarch Computer Architecture News*, *37*, 37–48.

Smaragdakis, Y., Evans, J., Sadowski, C., Yi, J., & Flanagan, C. (2012). Sound predictive race detection in polynomial time. *ACM Sigplan Notices*, *47*, 387–400.

Smith, S.O. (2013). *Raft: automated techniques for diagnosing, reproducing, and fixing concurrency bugs*. Ph.D. thesis, University of Cambridge.

Tan, L., Liu, C., Li, Z., Wang, X., Zhou, Y., & Zhai, C. (2014). Bug characteristics in open source software. *Empirical Software Engineering*, *19*, 1665–1705.

Tian, Z., Liu, T., ZHENG, Q., Zhuang, E., Fan, M., & Yang, Z. (2017). Reviving sequential program birthmarking for multithreaded software plagiarism detection. IEEE Transactions on Software Engineering (99), pp. 1–1.

Tillmann, N., & Halleux, J.D. (2008). Pex: white box test generation for .net. In *TAP'08 Proceedings of the 2nd International Conference on Tests and Proofs* (pp. 134–153).

Vaziri, M., Tip, F., & Dolby, J. (2006). Associating synchronization constraints with data in an object-oriented language. *ACM Sigplan Notices*, *41*, 334–345.

Veeraraghavan, K., Chen, P.M., Flinn, J., & Narayanasamy, S. (2011). Detecting and surviving data races using complementary schedules. In *Proceedings of the 23rd ACM symposium on operating systems principles* (pp. 369–384). ACM.

Voung, J.W., Jhala, R., & Lerner, S. (2007). Relay: static race detection on millions of lines of code. In *Joint meeting of the European software engineering conference and the ACM sigsoft international symposium on foundations of software Engineering* (pp. 205–214). Dubrovnik, Croatia.

Wang, C., Kundu, S., Limaye, R., Ganai, M., & Gupta, A. (2011). Symbolic predictive analysis for concurrent programs. *Formal Aspects of Computing*, *23*, 781–805.

Wang, C., Yang, Z., Kahlon, V., & Gupta, A. (2008). Peephole partial order reduction. In *Theory and practice of software, international conference on TOOLS and algorithms for the construction and analysis of systems* (pp. 382–396).

Wang, Y., Liao, H., Reveliotis, S., Kelly, T., Mahlke, S., & Lafortune, S. (2009). Gadara nets: Modeling and analyzing lock allocation for deadlock avoidance in multithreaded software. In *IEEE conference on decision and control* (pp. 4971–4976).

Wang, C., Limaye, R., Ganai, M., & Gupta, A. (2010). Trace-based symbolic analysis for atomicity violations. In *Proceedings of the 16th international conference on tools and algorithms for the construction and analysis of systems* (pp. 328–342).

Wang, H., Liu, T., Guan, X., Shen, C., Zheng, Q., & Yang, Z. (2017). Dependence guided symbolic execution. *IEEE Transactions on Software Engineering*, *43*(3), 252–271.

Weeratunge, D., Zhang, X., & Jaganathan, S. (2011). Accentuating the positive: atomicity inference and enforcement using correct executions. *ACM SIGPLAN Notices*, *46*, 19–34.

Xu, M., Bodík, R., & Hill, M.D. (2005). A serializability violation detector for shared-memory server programs. *ACM Sigplan Notices*, *40*, 1–14.

Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., & Bairavasundaram, L. (2011). How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on foundations of software Engineering* (pp. 26–36). ACM.

Yu, J., & Narayanasamy, S. (2009). A case for an interleaving constrained shared-memory multi-processor. *ACM Sigarch Computer Architecture News*, *37*, 325–336.

Yu, Y., Rodeheffer, T., & Chen, W. (2005). Racetrack: efficient detection of data race conditions via adaptive tracking. In *ACM SIGOPS operating systems review* (Vol. 39, pp. 221–234). ACM.

Yu, J., Narayanasamy, S., Pereira, C., & Pokam, G. (2012). Maple: a coverage-driven testing tool for multithreaded programs. *ACM Sigplan Notices*, *47*, 485–502.

Zhang, W., Sun, C., & Lu, S. (2010). Conmem: detecting severe concurrency bugs through an effect-oriented approach. *ACM Sigarch Computer Architecture News*, *38*, 179–192.

Zhang, W., Lim, J., Olichandran, R., Scherpelz, J., Jin, G., Lu, S., & Reps, T. (2011). Conseq: detecting concurrency bugs through sequential errors. *ACM Sigplan Notices*, *39*, 251–264.

Zhang, W., De Kruijf, M., Li, A., Lu, S., & Sankaralingam, K. (2013). Conair: featherweight concurrency bug recovery via single-threaded idempotent execution. *ACM SIGARCH Computer Architecture News*, *41*, 113–126.

Zhang, M., Wu, Y., Shan, L.U., Qi, S., Ren, J., & Zheng, W. (2016). A lightweight system for detecting and tolerating concurrency bugs. *IEEE Transactions on Software Engineering*, *42*(10), 899–917.

Haojie Fu is currently a master candidate in software engineering at Tianjin University, China. She received a B.Sc. degree in software engineering from Tianjin University, China, in 2015. Her research interests include software fault localization, concurrency debugging, and automatic program repair.

Zan Wang received a B.Sc. from the Department of Applied Mathematics (2000), a master's degree from the Department of Computer Science (2004), and a Ph.D. degree in Information Systems (2009) from Tianjin University, China, respectively. He is currently working as an associate professor at the School of Computer Software at Tianjin University. His research interests are mainly in software testing and analysis, such as software bug localization, concurrency bugs detection, and automatic repair.

Xiang Chen received a B.Sc. degree from the School of Management from Xi'an Jiaotong University, China, in 2002. He then received M.Sc. and Ph.D. degrees in computer science from Nanjing University, China, in 2008, and 2011, respectively. He is with the Department of Computer Science and Technology at Nantong University as an associate professor. His research interests are mainly in software testing, such as combinatorial testing, regression testing, and fault localization. He has published over 30 papers in referred journals or conferences.

Xiangyu Fan received a B.Sc. degree in software engineering from Tianjin University, China, in 2014. He is currently a master candidate in software engineering at Tianjin University, China. His research interests are mainly in software testing and software quality. He has some papers in referred journals about fault localization.