3-2020

IFIX: Fixing concurrency bugs while they are introduced

Zan WANG
Tianjin University

Haichi WANG
Tianjin University

Shuang LIU
Tianjin University

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Haoyu WANG
Tianjin University

See next page for additional authors

Citation

WANG, Zan; WANG, Haichi; LIU, Shuang; SUN, Jun; WANG, Haoyu; and CHEN, Junjie. IFIX: Fixing concurrency bugs while they are introduced. (2020). *2020 25th International Conference on Engineering of Complex Computer Systems (ICECCS): Singapore, March 4-6: Proceedings*. 155-164.
Available at: https://ink.library.smu.edu.sg/sis_research/5938

Author

Zan WANG, Haichi WANG, Shuang LIU, Jun SUN, Haoyu WANG, and Junjie CHEN

IFIX: Fixing Concurrency Bugs While They Are Introduced

Zan Wang
College of Intelligence and Computing
Tianjin University
Tianjin, China
wangzan@tju.edu.cn

Haichi Wang
College of Intelligence and Computing
Tianjin University
Tianjin, China
wanghaichi@tju.edu.cn

Shuang Liu*
College of Intelligence and Computing
Tianjin University
Tianjin, China
shuang.liu@tju.edu.cn

Jun Sun
School of Information Systems
Singapore Management University
Singapore
junsun@smu.edu.sg

Haoyu Wang
College of Intelligence and Computing
Tianjin University
Tianjin, China
wanghaoyu@tju.edu.cn

Junjie Chen
College of Intelligence and Computing
Tianjin University
Tianjin, China
junjiechen@tju.edu.cn

***Abstract*—Concurrency bugs are notoriously hard to identify and fix. A systematic way of avoiding concurrency bugs is to design and implement a locking policy that consistently guards all shared variables. Concurrency bugs thus can be viewed as the result of an illy-designed or poorly implemented locking policy. The trouble is that the locking policy is often not documented, which makes debugging concurrency bugs clueless. We argue that it is too late to debug concurrency bugs after programming is done and we instead detect and fix them while they are being implemented. In this work, we propose an approach named IFIX which flags potential concurrency bugs and recommends fixes while the bugs are introduced. The key idea is to automatically conjecture what the intended locking policy is based on static analysis and recommend fixes accordingly. The recommended fixes are present to the programmer promptly and the user feedback (i.e., whether the certain recommendation is selected) is used to refine the conjectured locking policy and consequently future fixes. IFIX is evaluated on 43 concurrent programs, and through a user study with 30 programmers. The experiment results and user feedback show that IFIX is efficient, accurate and user-friendly.**

***Keywords*—concurrency; bug fix; locking policy;**

I. Introduction

With the development of multi-core processors and the emerging requirements on high-performance tasks, multi-threaded programming is ubiquitous nowadays. Concurrency bugs (of multi-threaded programs) are known to be hard to debug and fix [1]. Firstly, it is challenging to identify and/or replay concurrency bugs due to the difficulty in manipulating thread interleaving. Secondly, fixing a concurrency bug is highly non-trivial due to the large number of steps and context switches in the test execution, most of which are irrelevant to the bug. Thirdly, it is challenging to conjecture a fix that avoids the bug with all possible thread interleaving.

There have been many proposals on identifying concurrency bugs (e.g., through testing [2], [3], [4], [5] or static analysis [6], [7]), replaying buggy traces [8], [9], understanding the cause of concurrency bugs [10], [11] and lastly fixing concurrency bugs [12], [13], [14], [15]. It is important to notice that existing approaches on fixing concurrency bugs are far from perfect. They focus on either atomicity violations [12], [16], [17], [18], deadlocks [19], [20], or data races [21]. Roughly speaking, these approaches design their fixes based on concrete execution traces which are either obtained from user-provided bug reports [12], [16] or runtime monitoring [18], [17], [21], or memory access patterns based on concrete execution traces [22], and provide no guarantee that the fix applies to unseen thread interleaving.

We argue that it is too late to debug concurrency bugs after programming is done. Instead, we propose to proactively identify and fix concurrency bugs while they are introduced. A key component of our approach is the locking policies, i.e., a function f from shared variables to locks such that $f(v) = l$ means lock l is consistently used to protect variable v. As stated by Peierls et al [23]., *the key to reduce concurrency bugs and ensure thread-safety is to design a locking policy according to the program specification*. That is, programmers ought to properly design a locking policy before implementing it so that concurrency bugs are avoided in the first place. We remark that a properly documented locking policy would be extremely helpful. For instance, concurrency bugs can be identified by checking if the locking policy is consistently implemented; the cause of a concurrency bug can be understood in terms of how a locking policy is ill-designed; and a concurrency bug can be fixed by either fixing the locking policy or the implementation if it deviates from the locking policy.

In this work, we propose IFIX which promotes the design and documentation of locking policies *without requiring a non-trivial amount of effort from programmers*. IFIX requires minimum user inputs and is designed to capture concurrency bugs when they are introduced. That is, IFIX silently scans the program for potential concurrency bugs based on static analysis. It automatically conjectures what the intended locking pol-

*Shuang Liu is the corresponding author.

978-1-7281-8558-3/20/$31.00 ©2020 IEEE
DOI 10.1109/ICECCS51672.2020.00025

icy is based on the program and generates recommended fixes accordingly. By law of parsimony, iFix always conjectures a locking policy that is the most consistent with the current implementation and thus requires minimum modification of the program. Furthermore, iFix learns from the programmer's selection to improve the accuracy of future recommendations. Lastly, the conjectured locking policy is automatically inserted into the program (in the form of annotations), which can be reviewed by the programmer and used for debugging concurrency bugs afterward if it is necessary.

iFix has been implemented as an Intellij IDEA plugin for Java programs and is open-source at **github.com/iFixConcurrency/iFix**. To evaluate its usefulness, we conduct a simulated experiment and a user study. In the simulated experiment, we collect a set of 43 concurrent programs with bugs and conduct multiple experiments to check whether iFix can be applied to eliminate the bugs. The results show that iFix detects concurrency bugs, generates the locking policy and recommends fixes correctly and efficiently. In the user study, we recruited 30 programmers to conduct non-trivial programming tasks and evaluate whether iFix is helpful in fixing concurrency bugs. All the programmers in the study agree that iFix captures mistakes timely, recommends fixes efficiently and correctly, and the generated locking policy always matches the intended one in their mind.

In summary, we make the following contributions. First, we propose a method to conjecture locking policies based on static analysis and MAX-SAT solving. Secondly, we develop a method and a tool iFix for preventing concurrency bugs when they are introduced. Thirdly, we conduct multiple experiments to show the effectiveness of using locking polices to prevent and fix concurrency bugs.

II. An Illustrative Example

In this section, we use an example to illustrate how iFix works step-by-step. Fig. 1 shows a concurrent program adopted from [24]. Object *mlst* is shared among all threads and if two builders (defined at line 25) call function *addLast* at the same time. Function *insert* is invoked by the two threads where *p._current* represents the same object. Consequently, there is a data race on a variable between line 7 and line 9 in the *MyLinkedList* class, i.e., multiple threads might access the same object *p._current* at the same time. To avoid the data race, line 7 and line 9 should be executed atomically. Furthermore, there is a subtle race between these lines and line 15 as well, where *p._current.next* is accessed through *itr.next* (since *p._current* and *itr* could be alias). Therefore, a consistent locking policy must be implemented to protect the shared object consistently throughout the program.

Given the program, iFix performs standard aliasing analysis, based on facilities offered by the D4 framework [25] for each shared variable. Afterward, a static happens-before graph is systematically built which allows us to detect data races. In this example, *p._current._next* is accessed at line 7 and 9, whereas *itr._next* is accessed at line 15. Aliasing analysis shows that *p._current* and *itr* are potentially alias and thus these three lines race with each other.

```
1 public class MyLinkedList {
2   ...
3   public void insert(Object x, MyLinkedListItr p) {
4     if (p != null && p._current != null) {
5       MyListNode tmp;
6       synchronized (this) {
7         tmp = new MyListNode(x, p._current._next);
8 //-       } commented by iFix
9         p._current._next = tmp;
10      } // + add by iFix
11    }}
12  public void addLast(Object x) {
13    MyListNode itr = this._header;
14    synchronized(this){ // + add by iFix
15      while (itr._next != null) itr = itr._next;
16     } // + add by iFix
17    insert(x, new MyLinkedListItr(itr));}}
18 class MyLinkedListItr{
19  public MyListNode _current; // Current position
20   MyLinkedListItr(MyListNode theNode){
21    this._current = theNode;}}
22 public class Main{
23   public static void main(String[] args){
24    Thread[] threads = new Thread[builders];
25    MyLinkedList mlst = new MyLinkedList(maxsize);
26    MyListBuilder mlistBuilder = null;
27    for (int i = 0; i < builders; i++) {
28       mlistBuilder = new MyListBuilder(
29         mlst,i*step,(i+1)*step,true);
30      new Thread(mlistBuilder).start();}}}
31 public class MyListBuilder implements Runnable{
32   public void run(){
33      for ( int i = this._bound1; i < this._bound2 ;i++
            ){
34        ((MyLinkedList)_list).addLast(new
              Integer(i));}}}
```

Fig. 1: An illustrative example

Once data races are identified, iFix checks whether there is certain locking policy that the programmer attempts to implement (but fails to do so correctly). In this example, since the race is on variable *p._current._next*, iFix identifies all locks that are used to protect *p._current._next* through static analysis. In particular, iFix traverses through the static happens-before graph to generate operation sequences on variable *p._current._next*. For instance, the following is an operation sequence which represents line 6 to 8.

$$\begin{aligned}&(lock(this), 6), (read(p.\_current.\_next), 7)\\&(write(tmp), 7), (unlock(this), 8)\end{aligned} \quad (1)$$

where each event in the sequence represents an access of a variable or a locking/unlocking event at a certain line. The above sequence of operations shows that (at least sometimes) *p._current._next* is protected with a lock on *this*.

Based on the above analysis results, iFix then generates a set of constraints that capture the information on the currently implemented locking policy based on two rules. First, if a variable x is protected by a lock l in some operation sequence, we generate a constraint $lock(x) = l$. Second, if variable x and y are protected by the same lock, we generate a constraint $lock(x) = lock(y)$. In this example, the constraints generated based on the above-mentioned operation sequence are $lock(\_next) = this$.

After obtaining all constraints, iFix conjecture the intended locking policy based on the principle of Occam's razor, i.e., the intended locking policy should be minimally different from what has been programmed. A locking policy should protect each shared variable (e.g., *p._current._next*) with a lock. In this example, the lock could be either *this* or a freshly created lock say $\mu lock$. Using MAX-SAT solving techniques, we then identify a locking policy that maximally satisfies the above-collected constraints. In this example, protecting *p._current._next* with a lock on *this* is a better locking policy since it satisfies all of the constraints, whereas protecting it with $\mu lock$ satisfies none of the constraints.

We remark that constraint solving in this example is straightforward as there is only one shared variable. In general, there may be multiple shared variables and locks and thus constraint solving is non-trivial. After constraint solving, we retain the top-K locking policies which satisfy the most constraints and generate one fix recommendation in an interactive window for programmers to select. In this example, the conjectured locking policy is to protect *p._current._next* with a lock on *this* and the recommended fix is to expand the *synchronized* block at line 6 to include line 9 and enclose line 15 with a *synchronized* block. Figure 1 shows the program after iFix automatically applies the fix.

III. iFix

In this section, we present the details of iFix. There are four main steps. First, iFix conducts static analysis to detect data races. Secondly, iFix conjectures a locking policy that is minimally different from the current program. Thirdly, a fix recommendation is generated based on the conjectured locking policy. Lastly, iFix automatically applies the fix. In the following, we present details of each step.

A. Static Race Detection

iFix relies on D4 [25] for static race detection, which is a static analysis framework that can be used to detect data races. We briefly summarize the D4 concurrency bug detection techniques that are relevant to our bug fixing techniques. We refer readers to the D4 [26] for more details. Note that our main contribution is on interactively generating fixes based on conjectured locking policies.

In general, data races are detected based on constructing a point assignment graph (PAG) and a static happens-before (SHB) graph. The PAG provides a mapping from variable names to memory locations called point-to set (pts), i.e., a set of memory locations. Aliases can be identified systematically by comparing the pts. The SHB graph captures the happens-before relations between control locations in different threads. Fig. 2 shows the SHB graph constructed from the program shown in Fig. 1.

With the SHB graph, iFix systematically identifies pairs of memory-accessing statements that can be executed concurrently (i.e., there are no chains of happens-before relation between the two statements). With the PAG graph, iFix then determines whether such pairs of statements access the same memory location (and one of the statements writes to the memory location). In other words, data races are systematically identified. For instance, as shown in Fig. 2, line 7 and 15 may execute concurrently by the two threads since there is no happens-before relation between them. Furthermore, the PAG graph shows that *p._current._next* and *itr._next* may point-to the same memory address. As a result, iFix concludes that there is a data race between line 7 and 15.

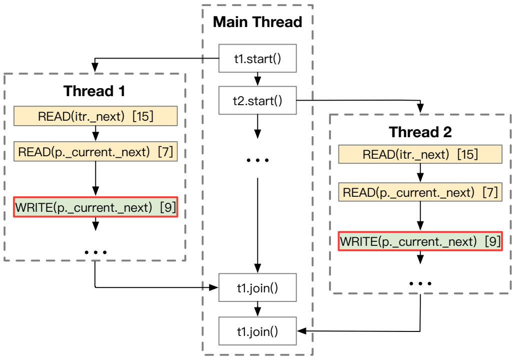

Fig. 2: Example SHB graph

Algorithm 1: Locking Policy Conjecture

```
initialize lp and lowerbound;
generatePolicy(lockList);
Function generatePolicy(lockList)
    calculate bestPossibleScore;
    if bestPossibleScore is smaller than lowerbound then
        return; // cut branch
    if A lockList is complete then
        calculate the score for lockList;
        if score is greater than lowerbound then
            update lowerbound and lp;
        return;
    for lock in Lock(x) where $x$ is the next variable do
        lockList.append(lock);
        generatePolicy(lockList);
        lockList.removeLast();
    return;
```

Note that a sequence of statements can be generated based on the SHB graph as the evidence of the data race. We remark that both the PAG graph and the SHB graph are constructed based on static analysis and thus the graphs could be not accurate, i.e., the PAG graph typically over-approximates whereas the SHB graph typically under-approximates (due to missing certain subtle happens-before relation). As a result, there might be false alarms or false negatives in the race detection results.

B. Locking Policy Conjecture

Once potential data races are identified, the next step is to generate fix recommendations. Existing approaches often introduce new locks (to protect the racing statements) [27], [16], [28], [29] and thus result in excessive locks when there are many races. A more systematic approach, as promoted in [30], is that *"to design a locking policy according to the* *program specification"*. That is, data races are the result of ill-designed/implemented locking policies. IFIX is designed to infer the intended locking policy automatically during the programming phase and gather user feedback based on the fix recommendations (e.g., which recommendation is selected) to refine the inference results. Furthermore, generating fix recommendations based on the inferred locking policy often allows us to reuse existing locks and avoid introducing redundant locks.

Formally, a locking policy is a function $lockP : V \to L$ where V is the set of shared variables and L is the set of locks. It is a function as a shared variable must be guarded by exactly one lock following [30]. We use $lock(x) = l$ to denote that variable x is protected by lock l. A locking policy is consistently implemented if and only if every access to variable x is guarded by a lock on l throughout the program.

Identifying the set of shared variables is straightforward based on the PAG graph and the SHB graph. To infer what is the intended locking policy, IFIX systematically analyzes the program to check whether there are existing protections by traversing the SHB graph and obtain operation sequences on the shared variables. An operation sequence is a sequence of read/write and lock/unlock operations. Based on the operation sequences, we can identify where locking and unlocking take place. For the example shown in Fig. 1, the operation sequence corresponding to line $6 - 10$ is shown in the list (1) in Section II, which suggests that the programmer intends to protect variable *p._current._next* with a lock on *this*.

Furthermore, given there are data races, the locks are likely not used consistently, i.e., some operation sequences may protect the same shared variable with different locks or no lock at all. To conjecture what the intended locking policy is, IFIX takes a global view of all operation sequences and gathers the information in the form of constraints. For every operation sequence with a pair of locking and unlocking, IFIX generates two kinds of constraints.

- Type I: If a variable x is accessed in between $lock(l)$ and $unlock(l)$, IFIX generates a constraint $lock(x) = l$, indicating that the programmer intends to protect x with lock l.
- Type II: If two variables x and y are accessed in between $lock(l)$ and $unlock(l)$, IFIX generates a constraint $lock(x) = lock(y)$, i.e., it seems possible that the two variables are related and the programmer intends to protect both variables using the same lock.

For example, given the operation sequence of line 6-10 in the original program (list (1) in Section II), we obtain constraint $lock(\_next) = this$ based on the first rule and $lock(p.\_current) = lock(p)$ based on the second rule. Note that because there is no race on variable p, the second constraint is discarded.

C. Locking Policy Generation

After the last step, we have collected a set of constraints that capture the existing locking policy. Since the current implementation is buggy, the existing locking policy is ill-designed/implemented, e.g., the same variable is not always protected by the same lock or related variables are not always protected by the same lock. For the example shown in Fig. 1, the reason for the data race is that line 7 and 9 both access the same variable whereas line 9 is not protected by lock $this$. The existing locking policy thus needs to be replaced with one which systematically protects the shared variables. Rather than generating a new locking policy from scratch, we aim to generate one which would fix the data races whilst being minimally different from the existing one.

```
method1() { synchronized (a) { a.update(); }}
method2() { synchronized (a) { a.update(); c.update(); }}
method3() { synchronized (b) { a.update(); b.update(); }}
method4() { synchronized (b) { b.update(); }}
method5() { synchronized (a) { c.update(); }}
```

Fig. 3: An example for lock policy generation

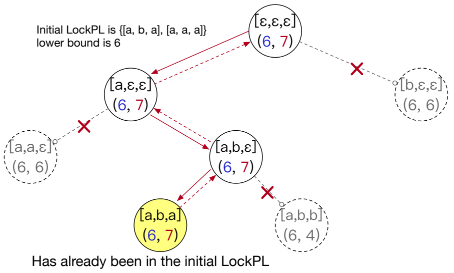

Fig. 4: An example illustrating Algorithm 1

We first identify a set of candidate locks for protecting each shared variable. For each variable x, if there is a constraint $lock(x) = l$, lock l is a natural candidate. Furthermore, any existing global objects are candidates as well (In Java, any reference-type object can serve as a lock). In addition, we assume that the candidate lock could be $\perp$, a special value denoting no protection, or a newly created fresh lock μ.

Next, we apply the principle of Occam's razor [31] to find a locking policy that maximally satisfies the constraints that we have collected, i.e., we apply MAX-SAT solving techniques to identify a locking policy that satisfies a maximum number of the collected constraints.

Algorithm 1 shows details on how IFIX generates the locking policy. It is a branch-and-bound algorithm for solving MAX-SAT problem which works reasonably efficiently in our setting. Note that we do not call existing MAX-SAT solvers due to the large overhead in starting and communicating with such solvers. The inputs are the set of shared variables *Var*, the set of candidate locks *Lock(x)* for each variable x, and the set of constraints *Con*. The output is a locking policy which satisfies the most number of constraints.

We maintain two variables, i.e., lp and $lowerbound$ which are the current best locking policy and the number of constraints that it satisfies. It is known that branch-and-bound works better if the initial "guess" is near-optimal. We thus initialize lp heuristically as follows. For each variable x, and

TABLE I: Satisfied constraints of program in Fig. 3

LP	Constraints						Satisfied		
	②	①	②	②	①	①	I	II	#Sum
	(a,a)	(a,b)	(b,b)	(c,a)	(a,c)	(a,b)			
[a,a,a]	Y			Y	Y	Y	4	2	6
[a,a,b]	Y					Y	2	1	3
[a,b,a]	Y		Y	Y	Y		6	1	7
[a,b,b]	Y		Y				4	0	4
[b,a,a]		Y		Y			3	0	3
[b,a,b]		Y			Y		1	1	2
[b,b,a]		Y	Y	Y		Y	5	1	6
[b,b,b]		Y	Y		Y	Y	3	2	5

each candidate lock l for x, we count the number of times a constraint of the form $lock(x) = l$ is in Con, i.e., the number of times l is used to protect x. lp is then set such that each variable x is to be protected by lock l such that $lock(x) = l$ occurs the most. Then *lowerbound* is set to the number of constraints satisfied by this lp.

A recursively defined function *generatePolicy* is then used to systematically enumerate locking policies. Assume there is a fixed order on the variables in Var. The parameter of the function $lockList$ is a partial locking policy, which is represented in the form of a sequence of locks where the first lock is assigned to the first variable, the second lock is assigned to the second variable and so on. Initially, $lockList$ is an empty list. Function *generatePolicy* then gradually completes $lockList$, one assignment at a time.

In particular, in line 5, we calculate the maximum number of constraints that are possibly satisfied by $lockList$, i.e., the number of constraints satisfied by the assigned variables in $lockList$ plus the number of constraints on the unassigned variables. If the result is smaller than $lowerbound$, the current $lockList$ is abandoned without ever completing it since it cannot be a locking policy better than lp. Otherwise, at line 7, if $lockList$ is complete (i.e., every variable is assigned with a lock), we check how many constraints that it satisfies and update $lowerbound$ and lp accordingly if $lockList$ is better than lp. Otherwise, the loop from line 12 to 15 tries assigning every candidate lock to the next unassigned variable and makes a recursive function call.

Note that the above generates the locking policy that satisfies the maximum number of constraints. To generate the top-K locking policies, a naive approach is to repeat the algorithm K times each time discard the previously identified one. A better approach is to amend Algorithm 1 slightly to record the top-K locking policies along the way. We skip the details for the sake of the presentation and instead discuss how it works with the example below.

In the following, we show how Algorithm 1 works using the example shown in Fig. 3, where a, b and c are three shared mutable objects and all the methods can possibly execute concurrently. Column 'Constraints' in Table I shows the constraints collected from the program. *(c,a)* represents the lock for variable c is a, and the circled number ② represents that the corresponding constraint appears twice, i.e., at line 2 and 5 of Fig. 3, respectively. Table I enumerates all the possible locking policies and the number of constraints satisfied by each locking policy. The first column shows the details of each locking policy (*[a,b,a]* means the corresponding lock for variable a, b and c respectively) and the last column shows how many constraints each locking policy satisfies.

Assume that we would like to identify the top-2 locking polices. We first initialize *LockPL* as $\{[a, b, a], [b, b, a]\}$ where $[a, b, a]$ denotes the locking policy $\{lock(a) = a, lock(b) = b, lock(c) = a\}$. These two locking policies are obtained by counting on the total number of constraints of type I. For instance, we initialize $lock(a) = a$ in the locking policy $[a, b, a]$ because there are a maximum of 4 constraints of the form $lock(a) = a$ as show in Table I; and a maximum of 2 $lock(b) = b$ and a maximum of 2 $lock(c) = a$. While this heuristic does not guarantee that the generated locking policy is optimal, it is often near optimal in practice. For instance, the locking policy $[a, b, a]$ turns out to be the optimal one in this example (see the third locking policy in Table I).

Fig. 4 shows the detailed step of Algorithm 1 for this example. Each node represents the current state, i.e., the value of $lockList$, $lowerbound$ (in blue) and the $bestPossibleScore$ (in red). Note that a value ϵ in $lockList$ means that no lock is assigned to the variable yet. The solid red arrows show the searching direction and the dashed red arrows show the backtracking direction. The grey colored nodes and edges are not explored due to line 5 and 6 in Algorithm 1. Initially, the top-2 locking policies are $\{[a, b, a], [b, b, a]\}$ and $lowerbound$ is 6, i.e., the number of constraints satisfied by $[b, b, a]$. Then, iFix calls function *generatePolicy* with parameters *lockList* $[\epsilon, \epsilon, \epsilon]$. First, *lockList* is set to $[a, \epsilon, \epsilon]$. After a recursive call, *lockList* is set to be $[a, a, \epsilon]$. Since the best possible score of this partial locking policy is 6, which is no larger than *lowerbound*. We backtrack and try $[a, b, \epsilon]$ next instead. Repeating the same steps for multiple times, we obtain the final top-2 locking policy as $[a, b, a]$ (satisfying 7 constraints) and $[a, a, a]$ (satisfying 6 constraints). This example shows that with a good "guess" on the initial bound, the branch-and-bound approach works efficiently.

D. Fix Recommendation and Application

After conjecturing the locking policy, iFix generates fix recommendations accordingly. For instance, for the example shown in Fig. 4, with the locking policy $[a, b, a]$, iFix recommends to additionally protect statement $a.update()$ in $method3()$ with a lock on a. The fix recommendations are shown in a popup window once the programmer clicks on the exclamation mark which highlights a data race in the program and will be applied automatically.

Automatically applying a fix requires us to refactor the program by introducing locking/unlocking statements at the right place. We may encounter two scenarios. One is that the statement is not protected by any locks. The other is that it is already protected by some locks. In the former case, iFix simply introduces a new *synchronized* statement. In the latter case, iFix checks whether the existing lock is consistent with the conjectured locking policy. If not, iFix first wraps the statement with a *synchronized* block with the intended lock and then checks whether the existing lock protects some other statements. For the example shown in Fig. 4, if the programmer selects the fix recommendation generated based on the locking policy $[a, a, a]$, IFIX first introduces a *synchronized(a)* block around the two statements in $method3()$. Afterward, it checks whether to remove $synchronzied(b)$ in $method3()$. According to the locking policy, lock b is no longer useful and thus the statement $synchronized(b)$ in $method3()$ is redundant and removed. Note that IFIX does not actually delete statement. Rather it comments out statements that are no longer needed.

IFIX also considers user feedback for improving future fix recommendations. When the user selects a certain fix, the choice is recorded and the locking policy preferred by the programmer previously will be given priority when new fix recommendations are generated. We remark that compared to existing approaches that fix programs after programming is done [28], [12], [16], [27], IFIX presents the fix recommendation timely during the programming phase. The advantage is that the intended locking policy is still fresh and thus the programmers can easily confirm whether our fix recommendation meets their expectation.

IV. Evaluation

IFIX is a self-contained toolkit built on top of existing program analysis tools for Java, including D4 [25], WALA [32], Akka [33] and Eclipse AST [34]. In particular, IFIX relies on D4, which is implemented based on WALA and Akka, to detect data races. Eclipse AST is a part of Eclipse JDT which is used to parse the Java programs into abstract syntax trees. IFIX is fully integrated with the Intellij IDEA IDE and is open-source at **github.com/iFixConcurrency/iFix** and has a total of $7,498$ lines of code. In the following, we systematically evaluate IFIX to answer four research questions (RQ).

- *RQ1: Is* IFIX *sufficiently efficient to provide instant feedback to programmers?*
- *RQ2: how accurate are the races detected by* IFIX*?*
- *RQ3: how accurate are the fix recommended by* IFIX*?*
- *RQ4: would real users recommend* IFIX*?*

A. Automated Experiments

To answer RQ1, RQ2, and RQ3, we conduct automated experiments on 43 programs collected from existing benchmarks including [35], [36], [24], [37], [38]. These programs are known to have concurrency bugs and are the subjects of various research. Relevant details of these programs are shown in Table II, where the second column shows the total number of non-comment-non-space lines of code in the program (without counting those in the invoked library).

To answer RQ1, we systematically apply IFIX to every program and measure the time taken by IFIX to identify the races. In particular, we measure the time taken to detect the bugs, the time to conjecture the locking policy (including constraint solving) and the time to apply the fix. IFIX is applied to each program 10 times and the average time is taken as the final measure. The results are shown in the fourth to sixth columns of Table II. All experiments are conducted on a PC with 16GB RAM, i7-6700 CPU, Windows 10 and JDK 1.8.0_191.

IFIX performs steadily time-wise to detect concurrency bugs, i.e., it takes about 2 seconds for most of the programs. One exception is the *JGFMolDyn* program, which takes around 9.3 seconds. The reason is that the program contains 51 shared variables and there are 144,621 pairs of potentially racing statements from different threads, which are time-consuming to check using the happens-before relations. The time taken for conjecturing the locking policy is negligible (i.e., less than 50 ms) for almost all the programs. The time for locking policy conjecturing mainly depends on the number of shared variables as well as the number of locks. The time for applying the fix is mostly negligible, although it has a wider range (i.e., from less than 50 milliseconds to around 1.8 seconds). The time varies because it mostly depends on the number of files to be modified. On average, it takes IFIX 2.410, 0.013 and 0.218 seconds to detect bug, conjecture locking policy and apply the fix, respectively.

Overall, the results show that *iFix* typically takes a few seconds to detect bugs and suggest fixes. Given that IFIX works in background and programmers often take a longer time to program, we believe such a delay is tolerable and thus *iFix* is sufficiently efficient.

To answer RQ2, we measure the number of bugs identified by IFIX as well as the number of false alarms for each program. For each program, we manually check the bugs reported to see whether it is an actual bug or it is a bug duplicating another or it is a false alarm. Note that two bugs are considered duplicate if they result in the same exception at the same line or they results in a race on the same variable by the same instructions.

The results are shown in column 'Detect Acc' of Table II. We can observe that in 34 (out of 43) cases, IFIX reports the bugs with 100% accuracy. In the remaining 9 cases, the number of false alarms varies from 1 to a maximum of 37 (in the case of *elevator*). A close investigation shows the false alarms are the result of imprecise static analysis. For instance, the pts may conservatively include variables which are not alias (e.g., newly initialized object) due to the limited precision of static aliasing analysis. Note that 2 programs have no data races (but rather bugs known as high-level races).

To answer RQ3, we manually check the correctness of the top-1 recommended fix for each actual bug in each program. A bug is considered fixed if the data race is successfully eliminated without introducing new bugs. Note that we use the actual fix for these programs as a golden standard. The results are shown in column 'Fix Acc' of Table II. As we can see from the results, IFIX correctly fixes all the detected bugs with its top-1 recommended fix. Furthermore, our fixes are generated based on conjecturing what is the intended locking policy and thus are consistent with existing implementations. For instance, for the example shown in Fig. 1, IFIX fixes the program by simply expanding an existing *synchronized* block at line 8 - 10, without introducing a new lock.

B. User Study

IFIX is designed for the programmers, i.e., offers recommendations and collects feedback. Thus, RQ4 can only be

TABLE II: Automated experiment results

program name	# LOC	Detect Acc	D time (s)	L time (s)	F time (s)	# Lock	Fix Acc
accountsubtype	129	1 / 1	2.002	0.005	0.493	5	1 / 1
airlinetickets	83	1 / 2	1.813	0.013	0.056	3	1 / 1
alarmclock	190	2 / 2	2.049	0.015	0.153	11	2 / 2
allocationvector	199	1 / 1	2.157	0.007	0.057	4	1 / 1
array	31	1 / 1	1.794	0.003	0.012	1	1 / 1
atmoerror	44	2 / 2	2.001	0.006	0.178	4	2 / 2
bakery	86	3 / 3	1.979	0.005	0.087	6	3 / 3
boundedbuffer	334	2 / 7	2.138	0.016	0.348	11	2 / 2
bubblesort	274	1 / 1	1.998	0.007	0.303	6	1 / 1
bufwriter	199	2 / 3	2.132	0.011	0.308	2	2 / 2
buggyprogram	161	2 / 2	1.843	0.005	0.040	3	2 / 2
bugsimplified	46	1 / 1	1.991	0.009	0.034	2	1 / 1
checkfield	39	1 / 1	2.196	0.009	0.048	2	1 / 1
consistency	28	1 / 1	1.987	0.003	0.052	1	1 / 1
critical	57	1 / 1	2.008	0.003	0.074	6	1 / 1
cyclicDemo	40	1 / 1	1.948	0.003	0.214	2	1 / 1
datarace	90	1 / 1	2.424	0.004	0.046	2	1 / 1
dekker	89	4 / 4	2.401	0.001	0.079	2	4 / 4
elevator	1155	7 /44	3.994	0.139	1.768	73	7 / 7
even	49	1 / 1	2.068	0.007	0.006	1	1 / 1
hashcodetest	987	2 / 2	2.084	0.012	0.172	3	2 / 2
JGFMolDyn	1010	3 / 8	9.329	0.055	1.290	17	3 / 3
JGFMonteCarlo	1478	0 /24	2.146	0.010	1.462	42	0 / 0
JGFRayTracer	1000	1 /11	2.427	0.023	1.099	29	1 / 1
lamport	126	5 / 5	5.291	0.044	0.074	4	5 / 5
linkedlist	175	1 / 1	2.332	0.009	0.006	2	1 / 1
mergesort	255	2 / 2	2.388	0.007	0.099	10	2 / 2
mix0	43	1 / 1	1.974	0.004	0.021	2	1 / 1
mix1	66	3 / 5	2.272	0.007	0.040	4	3 / 3
omcr	146	3 / 3	4.024	0.005	0.037	3	3 / 3
peterson	66	4 / 4	2.354	0.016	0.035	2	4 / 4
pingpong	117	1 / 1	2.241	0.006	0.032	3	1 / 1
pipeline	75	2 / 2	1.925	0.005	0.035	4	2 / 2
producerConsumer	134	1 / 1	2.108	0.003	0.024	2	1 / 1
rax	52	1 / 1	1.837	0.007	0.039	5	1 / 1
reorder1	67	1 / 1	2.014	0.010	0.017	2	1 / 1
sharedobject	45	1 / 1	2.039	0.006	0.024	2	1 / 1
store	43	1 / 1	1.816	0.003	0.006	1	1 / 1
stringbuffer	361	0 / 3	2.263	0.014	0.378	9	0 / 0
testArray	37	1 / 1	1.834	0.005	0.029	2	1 / 1
tso	33	1 / 1	1.656	0.026	0.038	2	1 / 1
wrongLock1	71	1 / 1	2.040	0.019	0.030	1	1 / 1
wrongLock2	36	1 / 1	2.308	0.008	0.052	1	1 / 1

* Detect Acc: the number of actual bugs / the number of reported bugs. D time is the detection time. L time is the time to conjecture the locking policy. F time is the time for fixing. Fix Acc: the number of correct fixes / the number of bugs.

TABLE III: Survey questions

ID	Question	Answer options
1	How difficult is the task?	1-5
2	How timely is the bug detection?	1-5
3	How accurate is the bug detection?	1-5
4	How user-friendly is the bug detection?	1-5
5	How helpful is the bug detection?	1-5
6*	How timely is the fix recommendation?	1-5
7*	How accurate is the fix recommendation?	1-5
8*	How user-friendly is the fix recommendation?	1-5
9*	How helpful is the fix recommendation?	1-5
10	Have you used static detection tools before?	Yes/No
11*	Have you used repairing tools before?	Yes/No
12*	Will you choose iFix (over other tools you used)?	Yes/No
13	How useful is iFix?	1-5
14	Any other comments or suggestions?	-

answered by programmers and thus a user study is conducted.

First, we identify a group of 30 volunteers through multiple channels (e.g., advertisement among students, researchers, and our industrial collaborators). The volunteers are then categorized into 3 levels based on their programming experience. L1 volunteers (18 in total) are undergraduate students with limited programing experience. L2 volunteers (6 in total) are postgraduate students with around one year of Java development experience. L3 volunteers (6 in total) are industry programmers or postgraduate students with more than 3 years of Java concurrency programming experience. The volunteers are randomly divided into two groups, the experiment group and the control group with a balance of levels.

Given the difficulty in identifying concurrency bugs, we develop a version of iFix (hereafter iFi) which only highlights the data races without providing any fix recommendations. Volunteers in the control group are provided with a tutorial on iFi. For each program, the volunteer starts with reading the program and clicks a button to highlight the data races. Afterward, the volunteer analyzes the program and fix the program. Volunteers in the experiment group are provided with a tutorial on iFix. The difference is they have recommended fixes after clicking the button. Afterward, the volunteer selects the recommended fix if he/she believes that the fix is correct, or modifies the program directly otherwise. There is no time limit for both groups. The time used for each stage of fixing and the fixing results are automatically recorded.

All volunteers are given three programs selected from the benchmark programs with different levels of difficulty. After finishing the task, the volunteers are required to fill in a questionnaire as shown in Table III. Both groups are asked whether the bug detection is efficient, accurate, useful and easy to use. The experiment group is asked the same set of questions on the bug fixing functionality. For questions $1-9$ and question 13, the options range from 1 to 5, where 1 means 'the least' and 5 means 'the most'. The result of the user study is summarized in Table IV. Column 4-6 are the total time each program takes on average. The last column shows whether the volunteer successfully fixes all three programs or not.

Considering that bug detection is an important step in iFix which has great impact on the overall user experience, and that the control group can only access the bug detection results, we design survey questions to check users' experience on bug detection and bug fixing separately.

Efficiency of iFix. We survey users' experience on the efficiency of bug detection (Q2) and bug fixing (Q6) separately. The survey results show that almost all volunteers agree that iFix both identifies bugs efficiently (average score 4.7 out of 5) and fixes bug efficiently (average score 4.7 out of 5). We further interview the volunteers who gave a score less than 4. Volunteer-1 rates 2 on question Q2, as he expects iFix to start working without requiring the programmer to click a button. The request is reasonable. However, there may be multiple entrances in a project and iFix would require additional knowledge on which entrance to start with.

Accuracy of iFix We survey users' experience on the accuracy of bug detection (Q3) and bug fixing (Q7) separately. Almost all volunteers agree that iFix detects bugs accurately (average score of 4.6) and fixes bugs accurately (average score of 4.4). Volunteer-7 rates 3 for Q3. He agrees that the

TABLE IV: User study results

group	ID	Level	P_A	P_B	P_C	Fix Result
			time / s	time / s	time / s	
A	1	1	522.60	488.39	786.63	Y
	2		543.81	423.45	909.51	Y
	3		455.18	539.56	613.77	Y
	4		311.63	349.09	499.32	Y
	5		287.17	703.00	1294.06	Y
	6		209.99	404.30	551.24	N
	7		748.38	1032.03	888.97	N
	8		445.54	455.17	927.85	Y
	9	2	223.45	244.98	566.36	Y
	10		459.12	483.35	538.52	Y
	11		234.40	336.34	428.90	Y
	12	3	241.05	255.00	503.94	Y
	13		395.18	361.59	461.16	Y
	14		206.93	320.73	497.47	Y
B	15	1	453.19	307.99	789.65	Y
	16		612.98	547.90	700.00	Y
	17		679.60	498.24	934.72	Y
	18		314.89	312.66	846.44	Y
	19		637.21	324.07	816.58	Y
	20		461.68	352.32	1394.49	Y
	21		338.19	300.40	480.17	Y
	22		361.20	286.23	1031.97	Y
	23		221.08	264.58	403.35	Y
	24		147.37	258.92	726.60	Y
	25	2	237.75	201.76	460.14	Y
	26		213.46	271.53	400.52	Y
	27		315.13	227.02	662.04	Y
	28	3	173.24	102.23	315.23	Y
	29		214.66	252.51	408.09	Y
	30		176.62	162.30	298.16	Y

[*] A is the control group; B is the experiment group. ID is the volunteer id. Level indicates the level (1-3) of the volunteer. Fix Results shows whether the volunteer successfully fixes all three programs or not.

detection results are accurate, but aruges that the information is insufficient to locate the bug. We remark that identifying the root cause of a bug is not the focus of this work. Volunteer-23 and 27 thought the fixs generated by iFix could be improved (and yet fail to demonstrate better fixes during the experiment).

Usefulness of iFix. Questions Q4 and Q8 ask whether iFix is user-friendly for bug detection (Q4) and for bug fixing (Q8). The survey results show that iFix is user-friendly in both bug detection (average score of 4.3) and bug fix (average score of 4.6). 6 volunteers (2 from control group and 4 from experiment group) give low rate on Q4. 5 of them suggest that iFix should start detection automatically without user clicking a button and the remaining one has concerns on whether all Java features like reflection are supported by iFix. Questions Q5 and Q9 are designed to check whether iFix is helpful in detecting and fixing bugs. The results show that iFix helps reduce the time to detect (average score of 4.5) or fix a bug (average score of 4.6). For Q5, volunteer-10 and 21 think the bug detection part of iFix is not useful as they found the bugs before clicking the button. This is because they take a long time to read and understand the program. For Q9, volunteer-21 and 29 think iFix did not reduce the time to fix bugs. In particular, volunteer-22 explains that he found the bugs before iFix is applied. Volunteer-27 responses even if a fix is automatically generated and applied, he still needs to check whether the fix is correct and does not introduce new bugs.

Comparing experiment group with control group Columns 4-6 show the the time spent on analyzing and fixing each program. The experiment group spent 5.79 minutes on average to fix Program-A which is 30 seconds (8%) less than the time spent by control group. For Program-B, experiment group spent 4.87 minutes on average to complete the job and is 2.75 minutes (36.11%) less than that spent by the control group. For the last program, The experiment group took 11.11 minutes on average and is 10 seconds (1.41%) less than the control group. For program-C, the average improvement is not very significant, this is because there are two volunteers (20 and 26) in the control group who take a long time to understand and analyse the program logic. They also take longer time to verify the fix patch provided by iFix. On average, the total time taken by the control group is 25.18 min, where that of the experiment group is 21.77 min. On average, the experiment group take about 3.4 minutes (13.5%) less than the control group to complete the task. The results show that iFix effectively reduces time for the program with a moderate difficulty level and not so much for the difficult program. One interpretation is that it takes much longer to understand the difficult program and validate the recommended fix. We further compare the time used for volunteers of different levels and find that the reduction in the total time (218.4, 175.35 and 380.01 seconds respectively for level-1, level-2, and level-3) are consistent across programmers at different levels.

Other findings. From the answers to Q1, we conclude that the volunteers' evaluation of the difficulty levels on the three programs is consistent with our evaluation. In particular, the easy, moderate, difficult program has a difficulty score of 1.63, 2.53 and 3.90, respectively. Questions Q10 and Q11 check the volunteers on whether they have prior experience of using similar tools and whether they would prefer iFix (than the other similar tools). The answers suggest there is a lack of similar tools in practice. Only volunteer-24, 28 and volunteer-30 have experience with concurrency bug detection tools and only volunteer-28 has used concurrency program repair tools. They both prefer to use iFix. The other volunteers also prefer to use iFix (than not using). For Q13, most volunteers give high ratings (average score of 4.48) for the usefulness of iFix.

The user study suggests that iFix is useful. All 16 volunteers in the experiment group successfully fix the bugs, whereas 2 Level 1 volunteers in the control group fail to fix the bug in the most difficult program. This seems to suggest that iFix is helpful for programmers with limited concurrency programming experience. In terms of the fixing quality, 3 volunteers in the control group add the *synchronized* keyword to every method in the program, which locks the entire method body. As a result, the degree of parallelism is significantly reduced and so is efficiency. On the contrary, most volunteers in the experiment group take the fixes recommended by iFix, which have a fine-granularity of locking, i.e., only statements accessing the relevant shared variables are protected. Volunteer 15, 19, 22 and 27 modify the applied fixes. They synchronize the whole method using a different lock. For instance, given program in Fig. 1, volunteer-15, 19, 22 fix the bug by synchronizing the whole `addLast` method and do not expand the lock scope at line $8-10$. Volunteer-27 keeps the changes at line $8-10$ but expands the lock scope to whole method (line $15-17$). Although all these 4 volunteers have fixed the bugs for the program, their fixes are less than ideal.

Threats to Validity Our evaluation may suffer from the following threats to validity. First, not all programs in our benchmark have clearly identified bugs and corresponding fix solutions. We run D4 on the fixed program and also conduct manual checking on the evaluation results to make sure the programs are fixed properly. Second, there are 43 programs in our benchmark, which are adopted from concurrency debugging and fixing related research. We tried our best to collect concurrency program benchmarks available. Experiments on larger projects could provide more confidence on the usability of IFIX. Last, the user study is conducted with 30 users of different programming experience. We study their survey results to evaluate the effectiveness and efficiency of IFIX. Although their fixing results and feedback on IFIX are consistent and representative, the number of volunteers could be further increased to make the results more convincing.

V. Related Work

Concurrency Bug Fixing. Approaches have been proposed to fix concurrency bugs effectively and efficiently. Approaches [28], [12], [16], [27] fix concurrency bugs by eliminating erroneous interleaving patterns. Huang *et al.* [27] propose to fix concurrency bugs by inserting synchronization. For fixing atomicity violation bugs, AFix [12] takes the CTrigger's [3] output as input and adds a mutex lock to the program to fix concurrency bugs. CFix [16] fixes concurrency bugs due to order violation based on AFix. CFix also enforces mutual exclusion with the same method. Axis [28] fixes atomicity violations by adding mutual exclusion locks and synchronization measures. Axis additionally takes efforts to reduce the possibility of introducing deadlocks. AlphaFixer [29] fixes atomicity violations by analyzing the lock acquisitions. It finetunes the locking so that it is possible to reduce the possibility of introducing deadlocks. HFix [13] fixes strategies guided by a survey of 77 manual patches of real-world concurrency bugs. HFix can also use the create and join operations of threads, while modifying the original locks to achieve the purpose of fix. PFix [22] proposes to fix concurrency bugs based on memory access patterns, which is the root cause of the concurrency bugs. PFIX is able to fix order violations, atomic violations, data races, which involve multiple variables.

These approaches on concurrency bug fixing do not consider interactive fixing during the programming phase, which is important for programmers to design correct locking policies. Our work proposes to fix concurrency bugs interactively during the programming phase, which enables us to provide instant feedback to programmers while the program design is still fresh in their minds.

Concurrency Bug Detection and Localization. Our work is related to the work on bug detection and localization. Extensive research has been conducted on localizing bugs with different strategies for both sequential programs [39], [40], [41], [42], [43], [44], [45] and multi-threaded programs [46], [5], [47], [48], [49]. CSight [46] generates a communicating finite state machine (CFSM) model by mining program execution logs. For race detection, IteRace [6] conducts static race detection in Java parallel loops. RaceMob [7] combines static and dynamic bug detection. During the static phase, it uses a static data race detector to find potential data races and updates a list of data races to developers. There are several approaches [2], [3], [4], [5] trying to expose concurrent bugs by inserting random disturbances, with the aim to increase the probability of triggering the rare interleaving executions where the bugs may be hidden. However, inserting random delay disturbance may cause high-performance overhead.

User Feedback Guided Debugging. There are several approaches which rely on user interaction to obtain feedback, most of the feedback is used to detect false alarms. BinGo [50] guides programs to find the true alarms leveraging user feedback. It converts Datalog derivation graphs into Bayesian networks and then computes alarm confidences based on feedback. Isil [51] presents a technique to help users classify error reports. It interacts with user with queries which capture missing facts. Users are required to answer those queries to provide feedback. Woosuk [52] clusters false alarms reported by static analyzers. Interactive techniques are used to reduce false alarms. URSA [53] uses user interaction to combine imprecise analysis with precise but unsound heuristics and then, it will pose questions to users to find the root cause. Different from previous works, our approach relies on static analysis techniques to detect data races and fix bugs interactively based on conjecturing the intended locking policy.

VI. Conclusion

We propose IFIX which facilitates interactive program fix on concurrency bugs during the programming phase. IFIX automatically detects data races and suggests fixes by conjecturing the intended locking policies. We conduct experiments with 43 concurrency programs and user studies on IFIX and the results show that IFIX is efficient, accurate and user-friendly. For future work, we plan to improve IFIX by supporting all Java features (including for instance, reflection) and other concurrency bugs such as high-level data races.

VII. Acknowledgement

This work was partially supported by the National Natural Science Foundation of China under Grant No. 61872263, U1836214 and 61802275, Key-Area Research and Development Program of Guangdong Province under Grant No. 2018B010107004, Intelligent Manufacturing Special Fund of Tianjin under Grant No. 20191012, 20193155, Innovation Research Project of Tianjin University under Grant No. 2020XZC-0042, 2020XRG-0022.

References

[1] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011, pp. 26–36.

[2] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A coverage-driven testing tool for multithreaded programs," *Acm Sigplan Notices*, vol. 47, pp. 485–502, 2012.

[3] S. Park, S. Lu, and Y. Zhou, "Ctrigger: exposing atomicity violation bugs from their hiding places," *Acm Sigplan Notices*, vol. 44, 2009.

[4] L. Chew and D. Lie, "Kivati: fast detection and prevention of atomicity violations," in *European Conference on Computer Systems, Proceedings of the European Conference on Computer Systems, EUROSYS 2010, Paris, France, April*, 2010, pp. 307–320.

[5] O. Inverso, T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato, "Lazy-cseq: A context-bounded model checking tool for multi-threaded c-programs," *Automated Software Engineering*, pp. 807–812, 2015.

[6] C. Radoi and D. Dig, "Effective techniques for static race detection in java parallel loops," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 4, pp. 24:1–24:30, September 2015.

[7] B. Kasikci, C. Zamfir, and G. Candea, "Racemob: Crowdsourced data race detection," in *Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 406–422.

[8] Y. Cai and W. K. Chan, "Magicfuzzer: scalable deadlock detection for large-scale applications," in *International Conference on Software Engineering*, 2012, pp. 606–616.

[9] Y. Cai, J. Zhang, L. Cao, and J. Liu, "A deployable sampling strategy for data race detection," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, 2016, pp. 810–821. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950310

[10] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now? an empirical study of bug characteristics in modern open source software," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, 2006.

[11] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008.

[12] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," *Acm Sigplan Notices*, vol. 46, pp. 389–400, 2011.

[13] H. Liu, Y. Chen, and S. Lu, "Understanding and generating high quality patches for concurrency bugs," in *The International Symposium on the Foundations of Software Engineering*, 2016.

[14] S. Khoshnood, M. Kusano, and C. Wang, "Concbugassist: Constraint solving for diagnosis and repair of concurrency bugs," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 165–176.

[15] P. Liu, O. Tripp, and C. Zhang, "Grail: context-aware fixing of concurrency bugs," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014.

[16] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu, "Automated concurrency-bug fixing," in *Usenix Conference on Operating Systems Design and Implementation*, 2012, pp. 221–236.

[17] P. Liu, J. Dolby, and C. Zhang, "Finding incorrect compositions of atomicity," *In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 158–168, 2013.

[18] Y. Cai, L. Cao, and J. Zhao, "Adaptively generating high quality fixes for atomicity violations," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 303–314.

[19] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. Mahlke, "The theory of deadlock avoidance via discrete control," *In ACM SIGPLAN Notices*, vol. 44, no. 1, pp. 252–263, 2009.

[20] Y. Cai and L. Cao, "Fixing deadlocks via lock pre-acquisitions," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 1109–1120.

[21] K. Sen, "Race directed random testing of concurrent programs," *ACM Sigplan Notices*, vol. 43, no. 6, pp. 11–21, 2008.

[22] H. Lin, Z. Wang, S. Liu, J. Sun, D. Zhang, and G. Wei, "Pfix: fixing concurrency bugs based on memory access patterns," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM, 2018*, 2018, pp. 589–600.

[23] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes, *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[24] "Software-artifact infrastructure repository," https://sir.csc.ncsu.edu.

[25] "D4," https://github.com/parasol-aser/D4, 2018.

[26] B. Liu and J. Huang, "D4: fast concurrency debugging with parallel differential analysis," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, 2018*, 2018, pp. 359–373.

[27] J. Huang and C. Zhang, "Execution privatization for scheduler-oblivious concurrent programs," in *ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2012.

[28] P. Liu and C. Zhang, "Axis: Automatically fixing atomicity violations through solving control constraints," in *Proceedings of the 34th international conference on software engineering*. IEEE Press, 2012.

[29] Y. Cai, L. Cao, and J. Zhao, "Adaptively generating high quality fixes for atomicity violations," *In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 303–314, 2017.

[30] B. Goetz, T. Peierls, D. Lea, J. Bloch, J. Bowbeer, and D. Holmes, *Java concurrency in practice*. Pearson Education, 2006.

[31] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth, "Occam's razor," *Readings in machine learning*, pp. 201–204, 1990.

[32] "Wala," https://github.com/wala/WALA, 2006.

[33] "Akka," https://akka.io/, 2011.

[34] "Abstract syntax tree," http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html, 2018.

[35] "The mcr github," https://github.com/parasol-aser/JMCR, 2016.

[36] "The rvpredict official website," http://fsl.cs.illinois.edu/rvpredict/, 2014.

[37] "Pecan benchmarks," http://www.cse.ust.hk/prism/pecan/\#Experiment.

[38] "The comrade github," https://github.com/buptsseGJ/ComRaDe, 2018.

[39] D. Hao, L. Zhang, T. Xie, H. Mei, and J.-S. Sun, "Interactive fault localization using test information," *Journal of Computer Science and Technology*, vol. 24, no. 5, pp. 962–974, 2009.

[40] D. Hao, T. Xie, L. Zhang, X. Wang, J. Sun, and H. Mei, "Test input reduction for result inspection to facilitate fault localization," *Automated Software Engineering Journal*, vol. 17, no. 1, pp. 5–31, March 2010.

[41] S. Wang, D. Lo, L. Jiang, Lucia, and H. C. Lau, "Search-based fault localization," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, no. 4. IEEE Computer Society, 2011, pp. 556–559.

[42] F. S. Ocariza Jr., K. Pattabiraman, and A. Mesbah, "Autoflox: An automatic fault localizer for client-side javascript," in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, no. 10. Washington, DC, USA: IEEE Computer Society, 2012, pp. 31–40.

[43] Q. Wang, C. Parnin, and A. Orso, "Evaluating the usefulness of ir-based fault localization techniques," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, no. 11. New York, NY, USA: ACM, 2015, pp. 1–11.

[44] X. Li, M. D'Amorim, and A. Orso, "Iterative user-driven fault localization," in *Hardware and Software: Verification and Testing: 12th International Haifa Verification Conference, HVC 2016, Haifa, Israel, November 14-17, 2016, Proceedings*, 2016.

[45] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2017.

[46] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with csight," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 468–479.

[47] B. Lucia, B. P. Wood, and L. Ceze, "Isolating and understanding concurrency errors using reconstructed execution fragments," *Acm Sigplan Notices*, vol. 46, pp. 378–388, 2011.

[48] S. Liu, G. Bai, J. Sun, and J. S. Dong, "Towards using concurrent java api correctly," in *Engineering of Complex Computer Systems (ICECCS), 2016 21st International Conference on*. IEEE, 2016, pp. 219–222.

[49] Z. Lin, H. Zhong, Y. Chen, and J. Zhao, "Lockpeeker: detecting latent locks in java apis," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 368–378.

[50] R. Mukund, K. Sulekha, H. Kihong, and N. Mayur, "User-guided program reasoning using bayesian inference," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, 2018*, 2018, pp. 722–735.

[51] I. Dillig, T. Dillig, and A. Aiken, "Automated error diagnosis using abductive inference," in *ACM SIGPLAN Notices*, vol. 47. ACM, 2012.

[52] W. Lee, W. Lee, and K. Yi, "Sound non-statistical clustering of static analysis alarms," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2012, pp. 299–314.

[53] X. Zhang, R. Grigore, X. Si, and M. Naik, "Effective interactive resolution of static analysis alarms," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 57, 2017.