Interleaving-tree Based Fine-grained Linearizability Fault Localization

Yang Chen[1,2], Zhenya Zhang[3⋆], Peng Wu[1,2] (✉), and Yu Zhang[1,2]

[1] State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
[2] University of Chinese Academy of Sciences
[3] National Institute of Informatics

Abstract. Linearizability is an important correctness criterion for concurrent objects. Existing work mainly focuses on linearizability verification of coarse-grained traces with operation invocations and responses only. However, when linearizability is violated, such coarse-grained traces do not provide sufficient information for reasoning about the underlying concurrent program faults. In this paper, we propose a notion of *critical data race sequence* (*CDRS*), based on our fine-grained trace model, to characterize concurrent program faults that cause violation of linearizability. We then develop a labeled tree model of interleaved program executions and show how to identify *CDRS*es and localize concurrent program faults automatically with a specific node-labeling mechanism. We also implemented a prototype tool, FGVT, for real-world Java concurrent programs. Experiments show that our localization technique is effective, i.e., all the *CDRS*es reported by FGVT indeed reveal the root causes of linearizability faults.

Keywords: Linearizability · Bug localization · Concurrency · Testing

1 Introduction

Localization of concurrency faults has been a hot topic for a long time. Multiple trials on a concurrent program with the same inputs may result in nondeterministic outputs. Hence, it is non-trivial to decide whether a concurrent program is buggy. Moreover, even if a concurrent program is known to contain a bug, it is difficult to reproduce the bug or to determine its root cause.

Efforts have been devoted to addressing the challenge of the localization of concurrency faults. The very basic way is to exhaustively explore the *thread schedule* space to replay and analyze the buggy executions. A thread schedule is usually described as a sequence of thread identifiers that reflects the order of thread executions and context switches. In [5, 26, 7], the thread schedule in a buggy execution is recorded and then replayed to reproduce the same bug. An execution of a concurrent program can be represented as a *fine-grained trace*,

⋆ The work was partially done when the author was a student at the Institute of Software, Chinese Academy of Sciences which is defined as a sequence of memory access instructions with respect to a specific thread schedule. In [14], fine-grained traces and correctness criteria are encoded as logical formulas to diagnose and repair concurrency bugs through model checking. Generally speaking, such fine-grained analysis suffers from the well-known state space explosion problem. Acceleration techniques have been presented to address this problem with, e.g., heuristic rules [2] or iterative context bounding [20]. However, most of these works aim at general concurrency faults, without cares about their nature or root causes.

In this paper, we focus on linearizability faults. Linearizability [12] is a widely accepted correctness criterion for concurrent data structures or concurrent objects. Intuitively, it means that every operation on a shared object appears to take effect instantaneously at some point, known as a *linearization point*, between the invocation and response of the operation, and the behavior exposed by the sequence of operations serialized at their linearization points must conform to the sequential specification of the shared object.

More attentions have been paid on linearizability verification recently [4, 3, 13, 17, 18]. In these works, an execution of a concurrent program is represented as a *coarse-grained trace*, which is defined as a partial order set of object methods. The basic approach of linearizability checking is to examine whether all the possible topologically sorted sequential traces satisfy the correctness criterion of the shared object. This approach also suffers from the state space explosion problem. Acceleration strategies have been proposed in these works to address this problem, too. However, since the coarse-grained trace model concerns only method invocations and responses, these techniques cannot determine the root causes of linearizability faults.

It is worth mentioning that data races and linearizability faults are different but related concepts. The occurrence of linearizability faults is due to the existence of data races. But not all the data races are critical to the linearizability faults. In this paper, we propose a notion called *Critical Data Race Sequence* (*CDRS*) based on the fine-grained trace model. Intuitively, a CDRS contains a sequence of data races that can decisively cause a linearizability fault in a concurrent program. Thus, the existence of a CDRS implies that the concurrent program can potentially produce a non-linearizable trace. In order to identify a CDRS, we model all the possible fine-grained traces of a concurrent execution as an *interleaving tree*, where each node corresponds to a data race, and each path from the root to a leaf node corresponds to a fine-grained trace. We label each node with a pre-defined symbol, depending on the linearizability of all the paths passing through the node. Then, the existence of a CDRS can be determined based on certain pattern of the node sequences in the labeled interleaving tree.

In order to overcome the state space explosion problem, we divide the localization process into two levels: the coarse-grained level and the fine-grained level. On the coarse-grained level, a *minimum* test case is to be synthesized that contains a sufficiently small number of operations to trigger a linearizability fault [29]. With such a small test case, the number of memory access instructions examined at the fine-grained level is greatly reduced. Together with the linearizability checking technique [18] and the coarse-grained linearizability fault localization technique [29], the overall localization process is shown in Fig.1, where stage C is concerned by this paper.

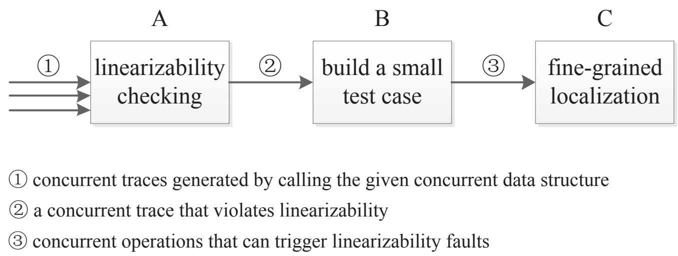

Fig. 1. Labels of nodes

For the brevity of presentation, Table 1 lists the abbreviations used throughout the paper.

Table 1. Abbreviations

CDRS	HLDR	CAS	FGVT
Critical **D**ara **R**ace **S**equence	**H**igh **L**evel **D**ata **R**ace	**C**ompare **A**nd **S**wap	**F**ine-**G**rained **V**eri**T**race

Contributions. The main contributions of this paper are as follows:

- We extend the traditional coarse-grained trace model to a *fine-grained trace* model by including memory access events. We also extend the notion of linearizability onto fine-grained traces.
- We propose the notion of *Critical Data Race Sequence* (CDRS), which plays a key role in characterizing the data races that can decisively result in linearizability faults.
- We develop a labeled *interleaving tree* model that contains all the possible fine-grained traces of a concurrent execution. Each node is marked automatically in a way that can reflect the existence of a CDRS through certain pattern.
- We implement a prototype tool, *FGVT*, for real-world Java concurrent programs. Experiments show that FGVT is rather effective in that all the CDRSes reported by FGVT indeed reveal the root causes of linearizability faults.

Related work. Automated linearizability checking algorithms have been presented in [4, 28], but exhibit a performance bottleneck. Based on [28], optimized algorithms have been proposed in [18] and [13] through partial order reduction and compositional reasoning, respectively. Model checking has been adopted in [3, 8] for linearizability checking with simplified first-order formulas, which can help improve efficiency. Fine-grained traces have been introduced in [17] to accelerate linearizability checking. These works lay a firm foundation for the localization of linearizability faults.

Active testing approaches have been proposed for concurrency bug localization based on bug patterns. The characteristics of bug patterns have been discussed in [19, 9] in details. Memory access patterns have been proposed in [23, 24, 16] for ranking bug locations. A fault comprehension technique has further been presented in [22] for bug patterns. Definition-use invariants have been presented in [25] for detecting concurrency bugs through pruning and ranking methods. A constraint-based symbolic analysis method has been proposed in [14] to diagnose concurrency bugs.

Some other concurrency bug localization techniques are based on the bug localization techniques for sequential programs. In [10], concurrent predicates are derived by an assertion mechanism to determine whether a data race causes a concurrency bug. Concurrent breakpoints, an adaption of the breakpoint mechanism, have been proposed in [21] for concurrent program debugging.

This paper aims at linearizability faults in fine-grained traces. It present a novel localization approach based on interleaving trees, which we propose for characterizing fine-grained traces in an effective manner. Hence, it opens a possibility to exploit the existing tree-based search techniques, e.g., partial order reduction, for efficient localization of the linearizability faults.

Organization. The rest of the paper is organized as follows. Section 2 presents an example to illustrate our motivation. Section 3 introduces our fine-grained trace model. Section 4 presents the key notion of CDRS based on our fine-grained trace model. Section 5 shows the labeled interleaving tree model, and the patterns of CDRSes. Section 6 reports the implementation and experiments about our prototype tool FGVT. Section 7 concludes the paper with future work.

2 Motivating Example

In this section, we illustrate the motivation of this work through a buggy concurrent data structure *PairSnapShot* [17].

Fig.2 shows a simplified version of PairSnapShot, where it holds an array `d` of size 2. A `write(i,v)` operation writes `v` to `d[i]`, while a `read` $\rightarrow \langle v_0, v_1 \rangle$ operation reads the values of `d[0]` and `d[1]`, which are v_0 and v_1, respectively.

A correctness criterion of PairSnapShot is that `read` should always return the values of the same moment. However, Fig. 3 shows a concurrent execution in which the return values of `read` do not exist at any moment of the execution. In Fig. 3, time moves from left to right; dark lines indicate the time intervals of operations and the short vertical lines at the both ends of a dark line represent the moment when an operation is invoked and returned, respectively. A label $t : \langle v_0, v_1 \rangle$ indicates that at the moment t, `d[0]` is v_0 and `d[1]` is v_1. The operation `read` on Thread 2 returns a value $\langle \mathtt{1}, \mathtt{2} \rangle$, which is not consistent with value of any moment.

The reason of this violation can be found out by enumerating the possible executing orders of memory access events which is labeled by `#` in Fig. 2. One possible order that can trigger the violation is illustrated in Fig. 4, in which "`x`" indicate the executing moments of the corresponding memory access events. Actually, this model checking approach is the most common way to locate the root cause of concurrency bugs, and has been studied in many existing literatures. Here, our focus is not on how to find this fine-grained executing order, but to study how the thread execution order, which causes data race, influences the final result of linearizability.

```
PairSnapShot:
    int d[2];
    write(i,v){
        d[i] = v;                #1
    }
    Pair read(){
        while(true){
            int x = d[0];        #2
            int y = d[1];        #3
            if(x == d[0])        #4
                return <x,y>;
        }
    }
```

Fig. 2. A concurrent data structure: PairSnapShot

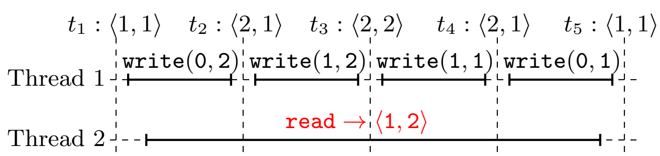

Fig. 3. A buggy trace of PairSnapShot

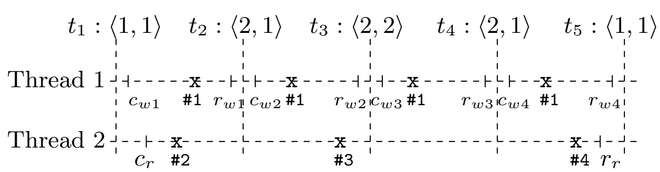

Fig. 4. An executing order of memory access events triggering the violation in Fig. 3

3 Preliminary

In this section, we extend the traditional coarse-grained trace model [3], recalled in Section 3.1, to the fine-grained trace model presented in Section 3.2. Compared to the traditional one, our new model includes memory access instructions such as `read`, `write` and atomic primitive *compare-and-swap* (`CAS`). This enables us to reason about the causes of linearizability faults on the fine-grained level.

3.1 Coarse-grained trace model

A *trace* S is a finite sequence of events $e(Arg)^{\langle o,t\rangle}$, where e is an event symbol ranging over a pre-defined set E, Arg represents the list of arguments, o belongs to a set O of operation identifiers and t belongs to a set T of thread identifiers. In the coarse-grained trace model, the set E contains the following subsets:

- C contains symbols that represent operation invocation events. An invocation event is represented as $c(v_a)^{\langle o,t\rangle}$ ($c \in \mathsf{C}$), where v_a is the argument of the operation;
- R contains symbols that represent operation response events. A response event is represented as $r(v_r)^{\langle o,t\rangle}$ ($r \in \mathsf{R}$), where v_r is the return value of the operation.

In this paper we also use C, R to represent the set of the corresponding events indiscriminately, and symbol $e \in \mathsf{C} \cup \mathsf{R}$ to represent an event. The order relation between events in a trace S is written as $\prec_S$ (or $\prec$), i.e., $e_1 \prec_S e_2$ if e_1 is ordered before e_2 in S. We denote the operation identifier of an event e as $\mathtt{op}(e)$, and thread identifier as $\mathtt{td}(e)$. An invocation event $c \in \mathsf{C}$ and a response event $r \in \mathsf{R}$ *match* if $\mathtt{op}(c) = \mathtt{op}(r)$, written as $c \diamond r$. A pair of matching events forms an operation instance with an operation identifier in O, and we usually represent such an operation instance as $m(v_a) \to v_r$, where m is the operation name.

A trace $S = e_1 e_2 \cdots e_n$ is *well-formed* if it satisfies that:

- Each response is preceded by a matching invocation:
  $e_j \in \mathsf{R}$ implies $e_i \diamond e_j$ for some $i < j$
- Each operation identifier is used in at most one invocation/response:
  $\mathtt{op}(e_i) = \mathtt{op}(e_j)$ and $i < j$ implies $e_i \diamond e_j$

A well-formed trace S can also be treated as a partial order set $\langle S, \sqsubset_S \rangle$ of operations on *happen-before* relation $\sqsubset_S$ between operations, where S is called a *coarse-grained trace* (or *coarse-grained history*). The *happen-before* relation $\sqsubset_S$ is defined as that: assuming two operations O_1, O_2 in S are formed by c_1, r_1 and c_2, r_2 respectively, then $O_1 \sqsubset_S O_2$ if and only if $r_1 \prec c_2$.

Example 1. Fig. 3 shows such a well-formed trace: $S = c_{w1} c_r r_{w1} c_{w2} r_{w2} c_{w3} r_{w3} c_{w4} r_r r_{w4}$, where c_{wi} and r_{wi} represents the invocation and response events of the i-th `write` operation respectively, and c_r and r_r represent the invocation and response events of the `read` operation.

In this example, it is obvious that $\mathtt{write}(0,2) \sqsubset_S \mathtt{write}(1,2) \sqsubset_S \mathtt{write}(1,1) \sqsubset_S \mathtt{write}(0,1)$, but there is no *happen-before* relation between the `read` operation and any of the `write` operations.

A coarse-grained trace S is *sequential* if $\sqsubset_S$ is a total order. We define that a *specification* of an object is the set of all sequential traces that satisfy the correctness criteria of that object. Note that here correctness criterion is specified by concurrent data structures, such as *first-in-first-out* rule of *FIFO-Queue*, *first-in-last-out* rule of *Stack*.

Definition 1 (Linearizability) *A coarse-grained trace S of an object is linearizable if there exists a sequential trace S' in the specification of the object such that:*

1. *$S = S'$, i.e., operations in S and S' are the same;*

2. $\sqsubset_S \subseteq \sqsubset_{S'}$*, i.e., given two operations* O_1 *and* O_2 *respectively in* S *and* S'*, if* $O_1 \sqsubset_S O_2$*, then* $O_1 \sqsubset_{S'} O_2$*.*

Note that this definition speaks only complete traces, neglecting the existence of pending operations, that is, the operations without response events. Since this paper focuses on analysis of linearizability faults rather than detection, we consider complete traces only.

Example 2. Fig. 5 shows 5 sequential traces that satisfy requirements 1 and 2 in Definition 1 with respect to the coarse-grained trace shown in Fig 3. However, neither of them satisfies the correctness criteria of *PairSnapShot*, by which the `read` operation should not return $\langle 1, 2\rangle$, so that neither of them belongs to the specification set of *PairSnapShot*. Therefore we can say that the coarse-grained trace in Fig. 3 is non-linearizable.

$$
\begin{array}{l}
1: \texttt{read} \rightarrow \langle 1,2\rangle \quad \texttt{write}(0,2) \quad \texttt{write}(1,2) \quad \texttt{write}(1,1) \quad \texttt{write}(0,1) \\
2: \texttt{write}(0,2) \quad \texttt{read} \rightarrow \langle 1,2\rangle \quad \texttt{write}(1,2) \quad \texttt{write}(1,1) \quad \texttt{write}(0,1) \\
3: \texttt{write}(0,2) \quad \texttt{write}(1,2) \quad \texttt{read} \rightarrow \langle 1,2\rangle \quad \texttt{write}(1,1) \quad \texttt{write}(0,1) \\
4: \texttt{write}(0,2) \quad \texttt{write}(1,2) \quad \texttt{write}(1,1) \quad \texttt{read} \rightarrow \langle 1,2\rangle \quad \texttt{write}(0,1) \\
5: \texttt{write}(0,2) \quad \texttt{write}(1,2) \quad \texttt{write}(1,1) \quad \texttt{write}(0,1) \quad \texttt{read} \rightarrow \langle 1,2\rangle
\end{array}
$$

Fig. 5. Five possible sequential traces but none of them satisfies the criterion of PairSnapShot

3.2 Fine-grained trace model

In the fine-grained trace model, the symbol set E has other subsets of events in addition to C and R:

- Wr contains symbols that represent the *memory writing* events. An event of memory writing is represented as $wr(addr, v)^{\langle o,t\rangle}$ ($wr \in \mathsf{Wr}$), where $addr$ is the memory location to be modified to a value v;
- Rd contains symbols that represent the *memory reading* events. An event of memory reading is represented as $rd(addr)^{\langle o,t\rangle}$ ($rd \in \mathsf{Rd}$), where $addr$ is the memory location to be read;
- CAS contains symbols that represent the *atomic primitive*, such as *compare-and-swap* (CAS) in modern architecture. A CAS event can be represented as $cas(addr, v_e, v_n)^{\langle o,t\rangle}$ ($cas \in \mathsf{CAS}$), where $addr$ represents a memory location, v_e and v_n are two values. The function of this atomic primitive is that if the value at $addr$ equals to v_e, it will be updated to v_n and return `true`, otherwise it would not do anything but return `false`.

Similarly, in this paper $\mathsf{Wr}, \mathsf{Rd}, \mathsf{CAS}$ also represent the corresponding sets of events. Let $\mathsf{M} = \mathsf{Wr} \cup \mathsf{Rd} \cup \mathsf{CAS}$, events e such that $e \in \mathsf{M}$ are called *memory access events*.

A fine-grained trace S_f is a total order set $\langle S_f, \prec\rangle$ of events over set $E = \mathsf{C} \cup \mathsf{R} \cup \mathsf{Wr} \cup \mathsf{Rd} \cup \mathsf{CAS}$. We define a projection $\mathcal{F}_c$ that maps a fine-grained trace S_f to a coarse-grained trace S_c by dropping all memory access events in S_f, i.e., $\mathcal{F}_c(S_f) = S_f|_{\{\mathsf{C},\mathsf{R}\}}$. A fine-grained trace S_f is *well-formed* if it satisfies that:

– $\mathcal{F}_c(S_f)$ is well-formed;
– Let $e_m(Arg)^{\langle o,t \rangle}$ be a memory access event in S_f. The thread identifier o of e_m must occur in a pair of matching invocation and response events $c(Arg)^{\langle o,t \rangle}, r(Arg)^{\langle o,t \rangle}$ in S_f.
– Let $c(Arg)^{\langle o,t \rangle}, r(Arg)^{\langle o,t \rangle}$ be a pair of matching invocation and response events in S_f. All the memory access events e with thread identifier o in S_f should satisfy $c \prec e \prec r$.

We claim that all fine-grained traces in this paper are well-formed.

Example 3. Fig.4 presents a well-formed fine-grained trace:
$S_f = c_{w1}c_r\texttt{\#2\#1}r_{w1}c_{w2}\texttt{\#1\#3}r_{w2}c_{w3}\texttt{\#1}r_{w3}c_{w4}\texttt{\#1\#4}r_r r_{w4}$
and the $\prec$ relations between events are obvious. Application of $\mathcal{F}_c$ to this fine-grained trace results in the coarse-grained trace in Fig.3.

Definition 2 (Linearizability of fine-grained trace) *The linearizability of S_f depends on the linearizability of $\mathcal{F}_c(S_f)$, i.e., if $\mathcal{F}_c(S_f)$ is linearizable, then S_f is linearizable.*

We define a predicate $\mathcal{L}_n$ to denote the linearizability of S_f, i.e., if S_f is linearizable, then $\mathcal{L}_n(S_f)$ is true.

4 Critical Data Race Sequence

In this section, we analyze linearizability faults on the fine-grained level, and propose *critical data race sequence* (*CDRS*) which exposes the root cause of linearizability faults.

Definition 3 (Concurrent program) *Given a coarse-grained trace S_c, we define a concurrent program $\mathbb{P}$: $\mathbb{P}$ is a set of fine-grained traces such that every fine-grained trace $S_f \in \mathbb{P}$ can be mapped to a coarse-grained trace S'_c, which satisfies that:*

– *$S'_c = S_c$, i.e., operations in S'_c and S_c are the same;*
– *$\sqsubset_{S_c} \subseteq \sqsubset_{S'_c}$, i.e., S'_c preserves the happen-before relation in S_c.*

Intuitively, a program $\mathbb{P}$ maintains all fine-grained traces that preserve the *happen-before* relation of S_c. If S_c is sequential, then we say the program $\mathbb{P}$ w.r.t. S_c is sequential. And, if there exists a non-linearizable fine-grained trace in a program $\mathbb{P}$, we say that $\mathbb{P}$ is non-linearizable.

Definition 4 (Linearizability fault) *Let $\mathbb{P}$ be a non-linearizable concurrent program. A linearizability fault $\mathfrak{F}$ is defined as a non-linearizable fine-grained trace S_f in $\mathbb{P}$.*

We define the prefix relation $\subseteq_{pre}$ between two fine-grained trace S_1 and S_2, that is, $S_1 \subseteq_{pre} S_2$ says that S_1 is a prefix of S_2. We use $\cdot$ to represent the concatenation of a sequence of events and another sequence of events, that is, if $S_1 = e_1 \cdots e_n$ and $S_2 = e_{n+1} \cdots e_{n+m}$, then $S_1 \cdot S_2 = e_1 \cdots e_n e_{n+1} \cdots e_{n+m}$.

Definition 5 (High-level data race [1]) *Let $\mathbb{P}$ be a concurrent program. A high-level data race (HLDR)* D *in* $\mathbb{P}$ *is defined as a triple* $\langle Var, I_e, M_e\rangle$*. Here, Var is a set of one or more shared variables, each corresponding to a memory location.* I_e *is a sequence of events.* M_e *is a set of two or more memory access events* $e(Arg)^{\langle o,t\rangle}$*, such that:*

- *each event e has a distinct thread identifier o;*
- *each event e accesses some shared variables in Var;*
- *for any permutation* S_p *of events in* M_e*, there exists a fine-grained trace* $S \in \mathbb{P}$ *such that* $I_e \cdot S_p \subseteq_{pre} S$.

Note that here the elements of *Var* depends on the algorithm of the object. The most common situation is that *Var* contains one shared variable which is accessed by several events simultaneously. However, there also exist other situations, such as *PairSnapShot* in Section 2, in which several memory locations should be considered globally.

Given a HLDR $\mathsf{D} = \langle Var, I_e, M_e\rangle$ where $e_1, e_2 \in M_e$, we say e_1 *wins* e_2 with respect to a fine-grained trace S_f if $I_e \cdot e_1 e_2 \subseteq_{pre} S_f$.

Given a program $\mathbb{P}$, we define a partial order relation $<_{dr}$ between two HLDRs $\mathsf{D}_1 = \langle Var_1, I_{e1}, M_{e1}\rangle$ and $\mathsf{D}_2 = \langle Var_2, I_{e2}, M_{e2}\rangle$ as that if $I_{e1} \subseteq_{pre} I_{e2}$, then $\mathsf{D}_1 <_{dr} \mathsf{D}_2$.

Theorem 1 *If there is a linearizability fault, then there is a high-level data race.*

Proof. To prove Theorem 1, it suffices to prove the contrapositive proposition that if there is no high-level data race, then there is no linearizability fault. According to Definition 5, the premise, no high-level data race, means that in M_e:

$$\exists S_p \forall S (I_e \cdot S_p \nsubseteq_{pre} S)$$

Here, if $\mathbb{P}$ is a concurrent program, this condition is not satisfied according to Definition 3. Therefore, the $\mathbb{P}$ that satisfies this condition corresponds to a sequential program, and the sequential trace surely has no linearizability fault.

Definition 6 (Critical data race sequence) *Let* $\mathbb{P}$ *be a concurrent program,* $\mathfrak{F}$ *be a linearizability fault. A Critical Data Race Sequence (CDRS) with respect to* $\mathfrak{F}$ *is a total order set of data races* $\{\mathsf{D}_1, \mathsf{D}_2, \cdots, \mathsf{D}_n\} =$

$$\left\{\begin{array}{c} \langle Var_1, I_{e1}, M_{e1} = \{e_{11}, e_{12}, \cdots, e_{1m_1}\}\rangle, \\ \langle Var_2, I_{e2}, M_{e2} = \{e_{21}, e_{22}, \cdots, e_{2m_2}\}\rangle, \\ \vdots \\ \langle Var_n, I_{en}, M_{en} = \{e_{n1}, e_{n2}, \cdots, e_{nm_n}\}\rangle \end{array}\right\}$$

where the relation $<_{dr}$ *exists as* $\mathsf{D}_1 <_{dr} \mathsf{D}_2 <_{dr} \cdots <_{dr} \mathsf{D}_n$*. A CDRS satisfies that there exist two events* $e_{i1}, e_{i2} \in M_{ei}$ $(i \in 1, \ldots, n)$ *that* e_{i1}*'s win and* e_{i2}*'s win lead the program to "inverse" consequences. Here, "inverse" includes two cases:*

- *If all the fine-grained traces S_{f1} satisfying $I_{ei} \cdot e_{i1} \subseteq_{pre} S_{f1}$ are linearizable, then there exist non-linearizable fine-grained traces S_{f2} satisfying $I_{e\,i} \cdot e_{i2} \subseteq_{pre} S_{f2}$;*
- *If all the fine-grained traces S_{f1} satisfying $I_{ei} \cdot e_{i1} \subseteq_{pre} S_{f1}$ are non-linearizable, then there exist linearizable fine-grained traces S_{f2} satisfying $I_{e\,i} \cdot e_{i2} \subseteq_{pre} S_{f2}$.*

Intuitively, a *CDRS* contains all the HLDRs which are decisive to the linearizability of the trace. Note that although different *CDRS*es in a program $\mathbb{P}$ may lead to different linearizability faults, what we focus on is just the linearizability of the trace and thus we treat all linearizability faults identical in the sense that they all lead the trace non-linearizable.

Example 4. Take a look at the example of a HLDR $\mathsf{D} = \langle Var, I_e, M_e \rangle$ in *PairSnapShot* in which $M_e = \{\texttt{\#1,\#2}\}$. If `#1` wins, then a non-linearizable trace will never occur; but if `#2` wins like Fig. 4, there exists at least one such fine-grained trace that is non-linearizable. In this case, D is included in a CDRS with respect to the linearizability fault $\mathfrak{F}$ shown in Fig. 4.

5 Identify CDRS on Interleaving Tree

From Section 4, we know that it is the competitions happening in *CDRS*es that trigger the linearizability faults. In order to identify *CDRS*, we propose an approach based on a model called labeled interleaving tree in this section. Firstly, we express the fine-grained traces in an interleaving tree, and then we label the nodes of the tree with a symbol system. We will show that all *CDRS*es follow a certain pattern and thus we can identify them based on the characteristics of the nodes.

5.1 Interleaving tree

Firstly, we define a projection $\mathcal{F}_M$ mapping a fine-grained trace S_f to a trace S_f^M composed of only memory access events in S_f, i.e., $\mathcal{F}_M(S_f) = S_f|_{\mathsf{M}}$. The linearizability of S_f^M is decided by that of S_f, i.e., $\mathcal{L}_n(S_f^M) = \mathcal{L}_n(S_f)$. We define a state $\mathcal{S}_t$ of an object to be a map from a memory locations to its value, e.g., in Fig. 3, $\mathcal{S}_t(\texttt{d}[0]) = 1$, $\mathcal{S}_t(\texttt{d}[1]) = 1$ at $\texttt{t}_1$.

Definition 7 (Interleaving Tree) *An Interleaving Tree of $\mathbb{P}$ is a tree, where each node corresponds to a state, and each edge corresponds to a memory access event. A subtree rooted at node N_d is represented as $\mathcal{T}ree(N_d)$. The set of the leaves of the tree is represented as N_{lf}.*

Algorithm 1 presents how to build an interleaving tree recursively. In line 1, $\mathcal{S}_t$ is initialized to the initial state of the object. The set enS in line 2 initially contains the events which are minimal w.r.t. $\prec$ over the events with the same thread identifier, and thus the number of elements in enS is the same as the number of threads. Line 3-11 is the process of building the tree. Firstly, a node is built as line 4 shows. Then, events in enS are traversed, each accessing a memory location $addr$ as line 5 shows. When an event is accessed, a corresponding edge is built as in line 6. Then the state is updated by substituting the value of $addr$ with v_n as in line 7. Here note that if e is an Rd event, $\mathcal{S}_t$ will not be modified. And enS is updated as line 8 shows, where e will be replaced by its successor w.r.t. $\prec$ over the events with the same thread identifier. Finally, the updated $\mathcal{S}_t$ and enS are applied as arguments to another invocation of BuildTree as line 9 shows to build a subtree recursively.

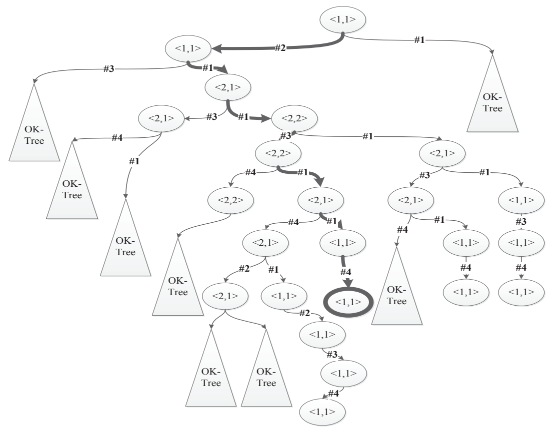

Fig. 6. Interleaving tree of PairSnapShot

Algorithm 1 Building of Interleaving Tree

1: $\mathcal{S}_t = \mathcal{S}_t^{init}$ ▷ $\mathcal{S}_t$ is initialized
2: $enS = \{e \mid \forall \epsilon \in \mathsf{M}.(\mathtt{td}(\epsilon) = \mathtt{td}(e) \longrightarrow e \prec \epsilon)\}$ ▷ Foremost events of each thread
3: **function** BuildTree($\mathcal{S}_t$, enS)
4: New Node($\mathcal{S}_t$)
5: **for** $e(addr, v_n)^{\langle o,t \rangle} \leftarrow enS$ **do**
6: New Edge(e)
7: $\mathcal{S}_t \leftarrow \mathcal{S}_t[v_n/addr]$ ▷ Update state
8: $enS \leftarrow enS \setminus \{e\} \cup \{e'\}$ ▷ Update enS
9: BuildTree($\mathcal{S}_t$, enS) ▷ Recursively build the tree
10: **end for**
11: **end function**

Example 5. According to Algorithm 1, we build the interleaving tree of the concurrent program $\mathbb{P}$ corresponding to Fig.3 and present it in Fig.6. Each node represents a state, and each edge represents a memory access event.

Although due to the limitation of space we have omitted many paths, we can still see that this tree maintains all the fine-grained traces in $\mathbb{P}$, and among these traces the one with bold paths corresponds to the non-linearizability situation in Fig.4.

5.2 Identify CDRS on labeled interleaving tree

After building an interleaving tree, we design a symbol system to label the tree in order for the identification of CDRS.

Since each leaf $l_f \in N_{lf}$ corresponds to a fine-grained trace from root to itself, we directly apply $\mathcal{L}_n$ to l_f to check the linearizability of the corresponding fine-grained trace. Here, we take a binary interleaving tree built by traces with two threads to illustrate our symbol system.

In our system, a subtree $\mathcal{T}ree(N_d)$ rooted at N_d and holding a leaf set N_{lf} can be grouped into one of the following categories:

- *OK*-tree — all fine-grained traces are linearizable, $\forall l_f\,(l_f \in N_{lf} \rightarrow \mathcal{L}_n(l_f))$
- *ERR*-tree — all fine-grained traces are non-linearizable, $\forall l_f\,(l_f \in N_{lf} \rightarrow \neg\mathcal{L}_n(l_f))$
- *MIX*-tree — both linearizable and non-linearizable fine-grained traces exist, $\exists l_{f1}(l_{f1} \in N_{lf} \wedge \mathcal{L}_n(l_{f1})) \wedge \exists l_{f2}(l_{f2} \in N_{lf} \wedge \neg\mathcal{L}_n(l_{f2}))$

Definition 8 (Node Labeling) *Based on the categories of subtrees, a node N_d can be labeled as one of the following symbols,*

- *W-node — if $\mathcal{T}ree(N_d)$ is an OK-tree.*
- *B-node — if $\mathcal{T}ree(N_d)$ is an ERR-tree.*
- *G-node — if one subtree of N_d is an OK-tree, and the other is an ERR-tree.*
- *GG-node — if two subtrees of N_d are both MIX-trees.*
- *WG-node — if one subtree of N_d is an OK-tree, and the other is a MIX-tree.*
- *BG-node — if one subtree of N_d is an ERR-tree, and the other is a MIX-tree.*

where W represents white, B represents black and G represents grey actually. Fig. 7 illustrates this labeling rule.

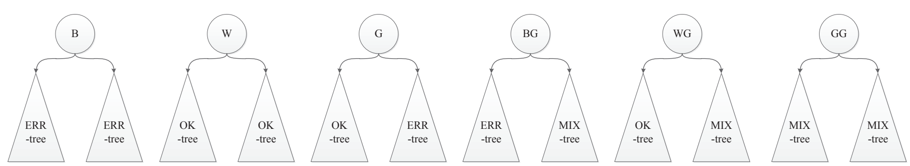

Fig. 7. Labels of nodes

The algorithm of labeling an interleaving tree is presented in Algorithm 2. The function LabelNode recursively labels the nodes of a tree. Firstly, it checks whether the node being labeled has left or right child in line {3,6,8,10}, where `Left`(N_d) gets the left child of N_d, and `Right`(N_d) gets the right child. `!Left`(N_d) means that N_d has no left child, and `!Right`(N_d) is in a similar way. So if both `!Left`(N_d) and `!Right`(N_d) are true, it means N_d is a leaf and thus it is labeled depending on the linearizability of itself as line 3-5 shows. Otherwise, the node is labeled depending its left and right child as line 6-27 shows.

Theorem 2 (Completeness) *Each node of the interleaving tree belongs to one kind of the nodes in Definition 8.*

Algorithm 2 Labeling Interleaving Tree

```
Label = {W, B, G, GG, WG, BG}
function LabelNode(N_d)
    if !Left(N_d)&!Right(N_d) then                      ▷ N_d is a leaf
        if L_n(N_d) then return W
        else return B
    else if Left(N_d)&!Right(N_d) then                  ▷ N_d has only left child
        return LabelNode(Left(N_d))
    else if !Left(N_d)&Right(N_d) then                  ▷ N_d has only right child
        return LabelNode(Right(N_d))
    else                                                ▷ N_d has both children
        Label_l = LabelNode(Left(N_d))
        Label_r = LabelNode(Right(N_d))
        switch ⟨Label_l, Label_r⟩                       ▷ The label depends on 2 children
          case ⟨W, W⟩:
            return W
          case ⟨B, B⟩:
            return B
          case ⟨B, W⟩|⟨W, B⟩
            return G
          case ⟨(B|W|G)?G, W⟩|⟨W, (B|W|G)?G⟩:
            return WG
          case ⟨(B|W|G)?G, B⟩|⟨B, (B|W|G)?G⟩:
            return BG
          case ⟨(B|W|G)?G, (B|W|G)?G⟩:
            return GG
        end switch
    end if
end function
```

Actually, each node N_d of an interleaving tree together with all of its out-edges corresponds to a data race $\mathsf{D} = \langle Var, I_e, M_e \rangle$. The set Var is a subset of the domain of $\mathcal{S}_t$, where $\mathcal{S}_t$ is represented by the value in a node N_d (e.g., Fig. 6), I_e is a prefix composed of events represented by edges from the root to N_d, and M_e contains all events e each corresponding to an out-edge of N_d. Therefore, we can uniquely identify a data race by a node.

Theorem 3 (Identifying CDRS) *A CDRS is equivalent to a subset of nodes in a root-to-leaf path, satisfying a regular expression form*

$$(W_g|B_g)^*(B_g|G)$$

where W_g, B_g, G respectively represent WG-node,BG-node,G-node.

Proof. – Firstly we show that the node sequence following $(W_g|B_g)^*(B_g|G)$ in an interleaving tree is a CDRS. From the definition of W_g-node, B_g-node, and G-node, it is obvious that the HLDRs composed by these 3 kinds of node and their out-edges all belong to the data races described in the Definition 6.

– Then we show that a CDRS appears as $(W_g|B_g)^*(B_g|G)$ in an interleaving tree. According to Definition 6, the two different cases for "inverse" consequences correspond to the W_g-node and B_g-node. Furthermore, since CDRS implies a linearizability fault $\mathfrak{F}$, the ending of a CDRS should be that there exists an event whose win can lead all fine-grained trace non-linearizable, and that is just the case of BG-node and G-node, which corresponds to the expression in the theorem.

Example 6. Take a look at Fig. 6. We label the tree according to Definition 8, and get a labeled tree in Fig. 8. As we can see, the thickened path with a red leaf is non-linearizable, the nodes on which include a CDRS. The CDRS is shown by the sequence of yellow nodes, in the form of $W_g\,W_g\,W_g\,W_g\,W_g\,G$, which is accepted by the regular expression in Theorem 3.

6 Implementation and Evaluation

We have integrated what we presented in Section 5 into a prototype tool called FGVT (Fine-grained VeriTrace), and experiments show that given a *minimum test case* [29], our tool is able to localize the CDRS. In this section, we will give a brief introduction about our tool and experiments, and display the experiment results to show the power of FGVT.

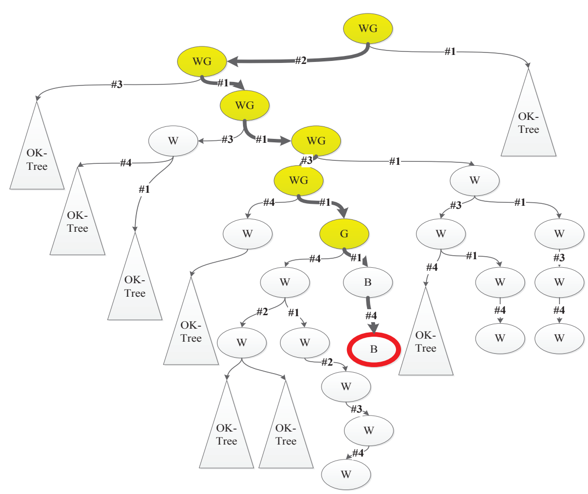

Fig. 8. Labeled interleaving tree

6.1 Implementation

Our tool FGVT is based on the framework of JavaPathFinder (JPF), which encapsulates a Java virtual machine and can be customized for use of model checking of Java programs. JPF is employed to generate interleaving trees, and it applies a dynamic reduction mechanism to eliminate duplicated program states and simplify the interleaving tree. Then, we label the tree based on Algorithm 2, and report the CDRSes which cause linearizability faults.

6.2 Benchmark

In this section, we will introduce some concurrent objects in our experiment. In addition to PairSnapShot, we also do experiments on many other concurrent objects, which have been proved to be non-linearizable by other tools.

Table 2. Evaluation Result

Concur. object	Initial State	Operations	CDRS	Relating data race
LockFreeList	{1}	thd1:`remove(1)` thd2:`remove(1)`	W_gG	`curr.next.get()` `attemptMark()`
OptimisticQueue	{1,2}	thd1:`poll()` thd2:`poll()`	G	`head.getItem()` `casHead()`

PairSnapshot	$\langle 1, 1\rangle$	thd1: `write(0,2)`,
		`write(1,2)`,
		`write(1,1)`,
		`write(0,1)`
		thd2:`read()`
			$W_g\{5\}G$	`d[i]=v`
				`x=d[0],y=d[1]`
				`if(x==d[0])`

Snark	{1}	thd1:`popRight()`
		thd2:`pushRight(2)`,
		`popLeft()`
			W_gW_gG	`rh=RightHat`
				`DCAS(&RightHat,...)`
				`DCAS(&LeftHat,...)`
				`if(rh.R==rh)`

SimpleList	{1}	thd1:`add(3)` thd2:`add(4)`	B_g or B_gG	`pred.next=node` `curr.val<v`
LinkedList	{1,5}	thd1:`remove(1)`, `add(9)` thd2:`size()`	W_gW_gG	`node.next==tail` `synchronized(){...}`

- LockFreeList [11] — It is a concurrent Set that violates linearizability when two `remove`s compete to mark a bit without synchronization protection.
- OptimisticQueue [15] — It is a concurrent Queue that violates linearizability when two `poll` operations compete to get the head of the queue. Without proper synchronization between reading `head` pointer and modifying it, two `poll`s may return the same value.
- Snark [6] — It is a Deque with the use of DCAS (*double-compare-and-swap*), and violates linearizability when the object has few elements and operations originally accessing different ends compete for the same memory location.
- SimpleList [27] — It is a concurrent Set and the bug is typical. The `Add` function inserts a node by modifying the `next` pointer of its predecessor, but without protection, `next` may be modified by other threads leading the node removed from the list unexpectedly.
- Operation `size` of Linked List — As we know, `size` is used for counting the number of nodes in a list. However, if there is no synchronization, a situation that violates linearizability happens when `size` traverses the list, another thread preempts the execution and deletes a node which has been accessed and inserts a node at a position that has not been accessed, so `size` will return a value that is larger than the expected length.

6.3 Evaluation

We evaluate our tool by 6 test cases either from prior work or from real applications as Section 6.2 introduced. In our experiments, all the concurrent objects are executed by two threads, and the operations being tested, initial states of each concurrent object and arguments are listed in Columns 2-3 of Table 2.

The node sequence patterns found based on the labeled interleaving tree are listed in Column 4 of Table 2. As we present before, these patterns exactly correspond to the CDRSes of each test case, and here we got some conclusions from the experimental results:

- All the patterns follow the form of regular expression in Theorem 3.
- Most of the test cases end with a G-node, and *SimpleList* shows us a sequence ending with a BG-node.
- We can see the case where not only one CDRS exists.

Column 5 lists the relating source code corresponding to the CDRSes. The source code is acquired from the events participating in the CDRSes, and facilitates the bug repair. For example, we can repair the linearizability faults in *LockFreeList* by transforming `attemptMark` into `compareAndSet`, while there also exist other situations, such as *PairSnapShot*, where we cannot point out exactly the modification of which instructions would lead the object linearizable, since all the data races participate in the CDRSes.

7 Conclusion

This paper proposes the notion of *critical data race sequence* (*CDRS*) that characterizes the root causes of linearizability faults based on a fine-grained trace model. A CDRS is a set of data races that are decisive to trigger linearizability faults. Therefore, the existence of a CDRS implies that a concurrent execution has potential to be non-linearizable. We also present a labeled interleaving tree model to support automated identification of CDRS. A tool called FGVT is then developed to automatically identify CDRSes and localize the causes of linearizability faults. Experiments have well demonstrated its effectiveness and efficiency.

This work reveals the pattern of the data races that are decisive on the linearizability of a trace. These data races can be mapped to certain parts of the source code. In the future work, it would be interesting to establish a stronger relationship between CDRSes and the source code for the sake of bug analysis and repair.

Acknowledgements

We thank the anonymous referees for their valuable comments. This work is partially supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST, the National Key Basic Research Program of China under the Grant No. 2014CB340701, the National Key Research and Development Program of China under the Grant No. 2017YFB0801900, and the CAS-INRIA Joint Research Program under the Grant No. GJHZ1844.

References

1. Artho, C., Havelund, K., Biere, A.: High-level data races. Softw. Test., Verif. Reliab. **13**(4), 207–227 (2003)
2. Ben-Asher, Y., Farchi, E., Eytani, Y.: Heuristics for finding concurrent bugs. In: International Symposium on Parallel and Distributed Processing. pp. 288a–288a (2003)
3. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Tractable refinement checking for concurrent objects. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. pp. 651–662. ACM (2015)
4. Burckhardt, S., Dern, C., Musuvathi, M., Tan, R.: Line-up: a complete and automatic linearizability checker. In: Zorn, B.G., Aiken, A. (eds.) Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010. pp. 330–340. ACM (2010)
5. Choi, J.D., Srinivasan, H.: Deterministic replay of java multithreaded applications. Proceedings of the Sigmetrics Symposium on Parallel & Distributed Tools pp. 48–59 (2000)
6. Doherty, S., Detlefs, D., Groves, L., Flood, C.H., Luchangco, V., Martin, P.A., Moir, M., Shavit, N., Jr., G.L.S.: DCAS is not a silver bullet for nonblocking algorithm design. In: Gibbons, P.B., Adler, M. (eds.) SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 27-30, 2004, Barcelona, Spain. pp. 216–224. ACM (2004)
7. Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby, G., Ur, S.: Framework for testing multi-threaded java programs. Concurrency and Computation: Practice and Experience **15**(3-5), 485–499 (2003)
8. Emmi, M., Enea, C., Hamza, J.: Monitoring refinement via symbolic reasoning. In: Grove, D., Blackburn, S. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015. pp. 260–269. ACM (2015)
9. Farchi, E., Nir, Y., Ur, S.: Concurrent bug patterns and how to test them. In: 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings. p. 286. IEEE Computer Society (2003)
10. Gottschlich, J.E., Pokam, G., Pereira, C., Wu, Y.: Concurrent predicates: A debugging technique for every parallel programmer. In: Fensch, C., O'Boyle, M.F.P., Seznec, A., Bodin, F. (eds.) Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, United Kingdom, September 7-11, 2013. pp. 331–340. IEEE Computer Society (2013)
11. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Elsevier (2012)
12. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
13. Horn, A., Kroening, D.: Faster linearizability checking via p-compositionality. In: Graf, S., Viswanathan, M. (eds.) Formal Techniques for Distributed Objects, Components, and Systems - 35th IFIP WG 6.1 International Conference, FORTE 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9039, pp. 50–65. Springer (2015)

14. Khoshnood, S., Kusano, M., Wang, C.: Concbugassist: constraint solving for diagnosis and repair of concurrency bugs. In: Young, M., Xie, T. (eds.) Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015. pp. 165–176. ACM (2015)
15. Ladan-Mozes, E., Shavit, N.: An optimistic approach to lock-free FIFO queues. In: Guerraoui, R. (ed.) Distributed Computing, 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3274, pp. 117–131. Springer (2004)
16. Liu, B., Qi, Z., Wang, B., Ma, R.: Pinso: Precise isolation of concurrency bugs via delta triaging. In: 30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014. pp. 201–210. IEEE Computer Society (2014)
17. Long, Z., Zhang, Y.: Checking linearizability with fine-grained traces. In: Ossowski, S. (ed.) Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016. pp. 1394–1400. ACM (2016)
18. Lowe, G.: Testing for linearizability. Concurrency and Computation: Practice and Experience **29**(4) (2017)
19. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: Eggers, S.J., Larus, J.R. (eds.) Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008. pp. 329–339. ACM (2008)
20. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: Ferrante, J., McKinley, K.S. (eds.) Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007. pp. 446–455. ACM (2007)
21. Park, C., Sen, K.: Concurrent breakpoints. In: Ramanujam, J., Sadayappan, P. (eds.) Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012. pp. 331–332. ACM (2012)
22. Park, S.: Fault comprehension for concurrent programs. In: Notkin, D., Cheng, B.H.C., Pohl, K. (eds.) 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013. pp. 1444–1446. IEEE Computer Society (2013)
23. Park, S., Vuduc, R.W., Harrold, M.J.: Falcon: fault localization in concurrent programs. In: Kramer, J., Bishop, J., Devanbu, P.T., Uchitel, S. (eds.) Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010. pp. 245–254. ACM (2010)
24. Park, S., Vuduc, R.W., Harrold, M.J.: A unified approach for localizing non-deadlock concurrency bugs. In: Antoniol, G., Bertolino, A., Labiche, Y. (eds.) Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012. pp. 51–60. IEEE Computer Society (2012)
25. Shi, Y., Park, S., Yin, Z., Lu, S., Zhou, Y., Chen, W., Zheng, W.: Do I use the wrong definition?: Defuse: definition-use invariants for detecting concurrency and sequential bugs. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA. pp. 160–174. ACM (2010)

26. Stoller, S.D.: Testing concurrent java programs using randomized scheduling. Electr. Notes Theor. Comput. Sci. **70**(4), 142–157 (2002)
27. Vechev, M.T., Yahav, E.: Deriving linearizable fine-grained concurrent objects. In: Gupta, R., Amarasinghe, S.P. (eds.) Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008. pp. 125–135. ACM (2008)
28. Wing, J.M., Gong, C.: Testing and verifying concurrent objects. J. Parallel Distrib. Comput. **17**(1-2), 164–182 (1993)
29. Zhang, Z., Wu, P., Zhang, Y.: Localization of linearizability faults on the coarse-grained level. In: He, X. (ed.) The 29th International Conference on Software Engineering and Knowledge Engineering, Wyndham Pittsburgh University Center, Pittsburgh, PA, USA, July 5-7, 2017. pp. 272–277. KSI Research Inc. and Knowledge Systems Institute Graduate School (2017)