Information and Software Technology 103 (2018) 162–173

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Diversity driven adaptive test generation for concurrent data structures

Linhai Ma[a,b,d], Peng Wu[*,a,b], Tsong Yueh Chen[c]

[a] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China
[b] University of Chinese Academy of Sciences, China
[c] Department of Computer Science and Software Engineering, Swinburne University of Technology, Australia
[d] Department of Computer Science, University of Miami, USA

A R T I C L E I N F O

Keywords:
Concurrent data structures
Test case diversity
Test case generation
Active testing
Adaptive random testing

A B S T R A C T

Context: Testing concurrent data structures remains a notoriously challenging task, due to the nondeterminism of multi-threaded tests and the exponential explosion on the number of thread schedules.
Objective: We propose an automated approach to generate a series of concurrent test cases in an adaptive manner, i.e., the next test cases are generated with the guarantee to discover the thread schedules that have not yet been activated by the previous test cases.
Method: Two diversity metrics are presented to induce such adaptive test cases from a static and a dynamic perspective, respectively. The static metric enforces the diversity in the program structures of the test cases; while the dynamic one enforces the diversity in their capabilities of exposing untested thread schedules. We implement three adaptive test generation approaches for C/C++ concurrent data structures, based on the state-of-the-art active testing engine Maple.
Results: We then report an empirical study with 9 real-world C/C++ concurrent data structures, which demonstrates the efficiency of our test generation approaches in terms of the number of thread schedules discovered, as well as the time and the number of tests required for testing a concurrent data structure.
Conclusion: Hence, by using diverse test cases derived from the static and dynamic perspectives, our adaptive test generation approaches can deliver a more efficient coverage of the thread schedules of the concurrent data structure under test.

1. Introduction

Concurrent data structures are key components for the development of concurrent software, because shared objects are often implemented with concurrent data structures. A concurrent data structure encapsulates self-synchronization mechanisms to coordinate the simultaneous accesses of multiple threads to a shared object. Presumably concurrent operations (e.g., method calls) of the shared object can be equivalently serialized to access the shared object sequentially. This assumption eases the development of concurrent software to a great extent, because developers just need to write sequential programs separately for individual threads without any reference to inter-thread synchronization. To be precise, the term *concurrent data structure* throughout the paper refers to a concurrent implementation of a data structure (e.g, a concurrent implementation of queue), instead of its sequential specification (e.g., a first-in-first-out sequential specification). Object-oriented programming languages have already provided typical concurrent data structures for multi-threaded programming, such as the *java.util.concurrent* package in Java. Plenty of open-source or proprietary concurrent data structures are also available to support the development of concurrent applications in practice.

Thus, the reliability of a concurrent data structure is vital to the correctness of a concurrent program that uses it. However, a concurrent data structure is no less error-prone than a concurrent program. Furthermore, testing a concurrent data structure may raise more challenges than testing a concurrent program because a concurrent data structure cannot run on its own. A test case of a concurrent data structure is a multi-threaded program that makes simultaneous accesses to the concurrent data structure. Obviously, the test case space is infinite in general. Therefore, it gives rise to a fundamental problem on the automated generation of effective multi-threaded test cases (i.e., concurrent programs) for concurrent data structures, let alone the challenge due to the nondeterministic execution of a test case and the exponential number of possible thread interleavings.

Active testing has established a coverage-guided paradigm for efficiently exploring the possible thread interleavings of a concurrent

* Corresponding author at: State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China.
E-mail addresses: l.ma@miami.edu (L. Ma), wp@ios.ac.cn (P. Wu), tychen@swin.edu.au (T.Y. Chen).

https://doi.org/10.1016/j.infsof.2018.07.001
Received 7 August 2017; Received in revised form 23 April 2018; Accepted 2 July 2018
Available online 04 July 2018

program under test. Typically, the state-of-the-art active testing tool Maple [1] works in two stages as follows: *profiling* and *active testing*. At the profiling stage, the concurrent program is profiled by running on its own, and the resulting concurrent executions are collected to discover the interleaving instances that encompass the shared-memory accesses by the multiple threads of the concurrent program. These include the interleaving instances that have taken place during the profiling executions, and the untested interleaving instances predicted based on the profiling executions. Then, at the active testing stage, the concurrent program is executed under purposely orchestrated thread schedules to expose the predicted interleaving instances. Such an *active test* terminates when no more interleaving instances can be exposed in the concurrent program. In this way, the behavior of the concurrent program under test can be exercised to a maximal extent, through a coverage saturation of the interleaving instances of the concurrent program.

In this paper, we propose an adaptive testing framework to tackle the test generation and execution problems in testing concurrent data structures, based on an active testing approach of concurrent programs. In this framework, a series of test cases are generated in a random but an *adaptive* manner in the sense that a successor test case is selected from a group of randomly constructed[1] candidates such that it is the most *diverse* one from all previously selected test cases in the context of interleaving instances. Then, such a diverse test case is executed for an active test, i.e., under purposely orchestrated thread schedules, to expose the interleaving instances between the operations of the concurrent data structure. The active test would terminate when no more interleaving instances are exposed in the concurrent data structure. In this way, the behavior of the concurrent data structure under test can be exercised to a maximal extent, through a coverage saturation of the interleaving instances of the concurrent data structure.

On one hand, diversity has been recognized as an important quality of test cases [2]. A test case space is usually infinite or too large to enumerate exhaustively. The notion of test case diversity suggests that evenly sampling the test case space can help avoid redundant tests that do not exercise any new behavior. In this paper, we propose to quantify the test case diversity for a concurrent data structure with respect to the interleaving instances between multiple operations of the concurrent data structure, based on the perspectives of the coarse-grained (operation) and the fine-grained (instruction) level.

On the other hand, following the principle of encapsulation in implementing concurrent data structures, a concurrent program usually can only interact with a concurrent data structure through the available operations supported by the concurrent data structure. Hence, in testing a concurrent data structure, we would focus on the interleaving instances that are induced only by the operations of the concurrent data structure. Through active testing of diverse multi-threaded test cases, which are generated adaptively for a concurrent data structure, our testing framework aims at reaching a saturation in covering the interleaving instances between multiple operations of the concurrent data structure, instead of the interleaving instances of the test cases themselves.

The main contributions of this paper are as follows. We present two diversity metrics that differentiate test cases for concurrent data structures from a static and a dynamic perspective, respectively. The *static* metric enforces the diversity in the program structures of test cases. Typically, a test case of a concurrent class consists of multiple sequences of method calls on a shared object of the class, while each sequence runs in a separate thread. The static diversity between the test cases can be measured by the distances between the method call sequences. Thus, this static metric characterizes the test case diversity from a coarse-grained perspective of operations. The *dynamic* metric enforces the diversity in the relative capabilities of test cases to expose interleaving instances on the instruction level. The dynamic diversity between the test cases can be measured with respect to the difference in the interleaving instances exposed by the previously executed test cases and the interleaving instances discovered by the newly generated test cases. Thus, this dynamic metric presents a fine-grained characterization of test case diversity, which can help avoid the redundancy in exposing the previously exposed interleaving instances.

Based on the above test case diversity metrics, we propose three adaptive test generation approaches for active testing of concurrent data structures. The three approaches generate a series of diverse test cases automatically, based on the static metric, the dynamic metric and their combination, respectively. In each round of test case generation, a finite number of candidate test cases are constructed randomly. Then, an optimal test case is selected among these candidates based on their relevant diversity measurements, and dispatched for active testing with the goal of an efficient exposition of the predicted interleaving instances of the concurrent data structure.

We then implement the framework for these three test generation approaches. Maple [1] is adapted as the active testing engine. The implementation is experimented with 9 open-source C/C++ concurrent data structures to evaluate its efficiency in detecting concurrency bugs. The results of the experiments show that, compared with purely random-based test generation, our approaches can expose more interleaving instances using less computation time and fewer tests.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 introduces the background and basic concepts used throughout this paper. Section 4 presents the definitions of the two test case diversity metrics, and the three adaptive test generation approaches are presented in Section 5. Section 6 reports the implementation of the adaptive testing framework and the results of the experiments. We discuss the limitations of this work in Section 7. The paper is concluded in Section 8 with future work.

2. Related work

Due to the vast variety of potential software faults, the notion of test case diversity is indeed the underlying principle of many test generation approaches [2], such as, partition testing [3], coverage testing [4], and adaptive random testing (ART) [5–7]. These approaches can generate diverse test cases spatially across the input domain or the structure of a program under test. For example, with diverse test cases, ART can find a program bug with up to 50% fewer test cases than random testing [6,8]. Our test generation approaches are inspired by the fixed-size candidate set ART technique (FSCS-ART) [5], which is the first and the most widely studied ART technique. FSCS-ART iteratively selects, from a fresh set of random candidates, a test input with the largest distance to the previously executed ones.

Efforts have been devoted to extend ART from numerical programs to non-numerical ones (e.g. [9]), and from sequential programs to concurrent ones [10]. A similarity metric has been proposed to evaluate distances between objects and method call sequences for object-oriented software [9]. We proposed an input-driven approach that can generate diverse test inputs for active testing of a multi-threaded program [10]. The test inputs are differentiated based on their capabilities in exposing the interleaving instances of the multi-threaded program on the instruction level.

In this paper, we further present a diversity-driven approach that can generate diverse multi-threaded programs for active testing of a concurrent data structure. The multi-threaded programs, used as test cases, are differentiated by their capabilities in exposing the interleaving instances of the concurrent data structure on both the operation and the instruction levels. Each test case typically consists of more than one method call sequence with regard to a shared object of the same type. But the similarity metric proposed in [9] is built for individual method call sequences over objects of various types, and hence it does not fit for the test cases of the concurrent data structure. Actually, not

[1] In this paper, random selection or generation refers to random selection or generation based on a uniform distribution.

all the interleaving instances of the test cases, but the interleaving instances between the operations of the concurrent data structure, affect the behavior of the concurrent data structure.

Few efforts have been devoted on automated test generation for C/C++ concurrent data structures. ConTeGe [11] generates randomly concurrent tests for Java classes, which unfortunately tend to repeat the already exercised behavior recurrently. CovCon [12] extends ConTeGe and can guide random test generation towards covering concurrent method pairs, which can be regarded as a lightweight abstraction of interleaving instances. Instead, the test case diversity metrics presented in this paper can lead to the coverage of interleaving instances on both the method and the instruction levels. A test case generation approach was presented in [13] for parallel software composed of loosely couple threads/processes, such as task trees. Their work is different to this study that aims at concurrent software composed of tightly coupled threads (due to shared memory).

Three categories of testing approaches have been developed for concurrent programs: *stress testing* [14], *systematic testing* [15–17] and *active testing* [1,18–22]. *Stress testing* runs a program under test repeatedly with a given input, but has small chances in exposing distinct interleaving instances [14]. Its performance may be improved by introducing random delays at synchronization points [23] or assigning random priorities to each thread [24]. *Systematic testing* explores all the possible interleaving instances under a given input, or within a bounded number of context switches. But, it inevitably encounters the scalability issue [15–17]. Dynamic partial-order reduction techniques have been adopted to reduce the search space [25,26]. *Active testing* first predicts suspicious interleaving instances with respect to certain types of concurrency bugs, e.g., races [18] and concurrency-memory bugs [19], and then enforces the potential buggy interleaving instances actively to reveal bugs. Coverage criteria [1,20–22] have been further proposed to determine whether the predicted interleaving instances are sufficiently enough to reveal concurrency bugs. The coverage metrics presented in [20,22] are based on synchronization primitives and specifically target at synchronization errors in concurrent programs. Noise injection [21] or orchestrated thread schedules [22] can then be applied to enhance the degree of coverage. The interleaving coverage metric defined by Maple [1] explicitly targets at interleaving patterns (idioms) between multiple threads. An interleaving idiom depicts a pattern of inter-thread dependencies, and characterizes a *happens-before* relation between conflicting memory accesses by different threads. For shared variables, two instructions are *conflicting* if both of them access the same variable and one of them is a write access. Conflicting synchronization instructions can be defined in a similar way [27]. Maple relies on dynamic binary instrumentation [28] to actively control thread schedules, which are targeted to expose the untested interleaving instances.

The active testing approach presents an efficient way to execute a multi-threaded program, while the coverage criteria present an effective measurement for evaluating the diversity of test inputs, as suggested by Yue et al. [10]. The testing framework presented in this paper focuses on adaptive generation of multi-threaded programs, which are to be executed with the active testing engine Maple [1] for exercising the behavior of a concurrent data structure under test. The coverage criteria are then lifted for evaluating the diversity of the multi-threaded programs. With minor effort, this adaptive testing framework can be implemented with other active testing engines.

3. Preliminaries

In this section, we briefly present some background, including the definition of a test case for a concurrent data structure [11], and the basic concepts of active testing [1]. Without loss of generality, let C be a concurrent class that represents a concurrent data structure under test.

Definition 1 (*Test case*). A *test case* of a concurrent class C is a tuple $t = (p, \{s_i | i \in [1, n]\})$, where $n > 1$ is the number of threads, p is a finite sequence of method calls to construct a shared object of class C, and s_i is a finite sequence of method calls of the shared object. For a test case t, p is called the *sequential prefix* of t, while s_i is called the i-th *concurrent suffix* of t for each $1 \le i \le n$.

The test case t essentially represents a concurrent program with n threads. Intuitively, when t is being executed, a shared object is first constructed through running its prefix p; then n threads are initiated, each assigned with a unique suffix s_i ($1 \le i \le n$) to access the shared object concurrently.

Definition 2 (*Interleaving instance*). An *interleaving instance* of a concurrent class is typically represented as $A_X \to B_X$, where A_X and B_X are two instructions accessing a shared variable X, respectively from two method calls of the concurrent class by distinct threads, and A_X happens before B_X.

Unless otherwise specified, the term *interleaving instance* refers to an interleaving instance of a concurrent class in the rest of this paper.

Definition 3 (*Basic concepts of active testing*). Consider a test case t, which is a multi-threaded concurrent program.

1. *Profiling the test case t* means that t is ran independently and non-deterministically, without any pre-determined thread schedule.
2. *Executing the test case t for an active test* means that t is ran with pre-determined schedules.
3. An interleaving instance is said to be *discovered* by a test case t in either of the following two ways:
   - If a pair (A_X, B_X) of instructions, made by two distinct threads accessing a shared variable X, occurs in a profiling execution of the test case t, and A_X happens before B_X, then the interleaving instance $A_X \to B_X$ is said to be *actually* discovered.
   - If A_X does not happen before B_X in any profiling execution of the test case t, where the pair (A_X, B_X) occurs, then the interleaving instance $A_X \to B_X$ is said to be *predictively* discovered.
4. An interleaving instance is said to be *exposed or tested* by a test case t in either of the following two ways:
   - Suppose an interleaving instance is actually discovered in a profiling execution of the test case t. Then, the interleaving instance is said to be exposed by directly profiling the test case t. In this scenario, the interleaving instance is said to be discovered and exposed by t.
   - Suppose an interleaving instance is predictively discovered from a profiling execution of the test case t (in this scenario, the interleaving instance is said to be discovered by t but not yet exposed). Then, the interleaving instance is to be exposed by executing the test case t for an active test, with a thread schedule determined according to the interleaving instance.

The following notations would be used throughout the paper:

- $iRoots(t)$ denotes the set of the interleaving instances that are discovered by a test case t.
- $iRoots(T)$ denotes the set of the interleaving instances that are discovered by the test cases in a set T, i.e.,
  $iRoots(T) = \bigcup_{t \in T} iRoots(t)$.
- $IRoots(t)$ denotes the set of the interleaving instances that are exposed by a test case t.
- $IRoots(T)$ denotes the set of the interleaving instances that are exposed by the test cases in a set T, i.e.,
  $IRoots(T) = \bigcup_{t \in T} IRoots(t)$.
  Obviously, $IRoots(T) \subseteq iRoots(T)$, because all the exposed interleaving instances must have to be discovered first.
- $|S|$ denotes the size of a finite set S, i.e., the number of elements in S.

4. Test cases diversity metrics

In this section, we present two diversity metrics for test cases of concurrent data structures.

4.1. Static metric

Syntactically, a finite sequence of method calls can be regarded as a character string of finite length. Edit distances [29] are common metrics for quantifying the difference between two character strings. For example, Levenshtein distance [30] is a typical edit distance metric in information theory and computer science. It is defined as the minimum number of edits required to change one string to the other, including insertions, deletions and substitutions. The Levenshtein distance between two character strings can be computed efficiently with the Wagner-Fischer Algorithm [31]. Thus, we can statically differentiate test cases based on the Levenshtein distances between their sequential prefixes and between their concurrent suffixes, as formally defined below.

Definition 4 (*Static diversity metric*). Let $d(t_1, t_2)$ denote the *static distance* between test cases $t_1 = (p_1, \{s_{1i} | i \in [1, n]\})$ and $t_2 = (p_2, \{s_{2i} | i \in [1, n]\}$.
Then, $d(t_1, t_2) = lev(p_1, p_2) + \sum_{i=1}^{n} lev(s_{1i}, s_{2i})$ where $lev(s, s')$ is the Levenshtein distance between sequences s and s'. For a test case t and a test case set T such that $t \notin T$, the static distance between t and T, denoted as $\Delta(t, T)$, is the minimum static distance between t and any test case in T, i.e., $\Delta(t, T) = \min_{t' \in T} d(t, t')$.

Herein, we presume that each randomly generated test case candidate has the same number of threads. The static distance $d(t_1, t_2)$ quantitatively characterizes the structural diversity between test cases t_1 and t_2 in a straightforward way, i.e., both test case differ in the possible concurrent executions of the class C's methods rendered by the test cases.

4.2. Dynamic metric

The static metric exhibits the test case diversity on the method level of a concurrent class, while the dynamic metric aims at the instruction level, i.e., the interleaving instances induced by the method calls of the concurrent class.

We can differentiate test cases dynamically based on the interleaving instances that can be discovered by the test cases. However, as testing of a concurrent class proceeds, its interleaving instances will be gradually exposed. Then, although a test case t can discover absolutely more interleaving instances than another test case t', t may discover only the interleaving instances that have already been exposed before, while t' may discover some new interleaving instances that have not yet been exposed. Thus, t' can improve the coverage of interleaving instances, but t cannot. Therefore, an absolute metric may offset the intuition of test case diversity if it opts for t instead of t'. Hence, we are going to present a relative metric, which takes the previously exposed interleaving instances into account.

Note that, in active testing, interleaving instances are discovered through profiling a test case. Some of the discovered interleaving instances indeed take place in the profiling executions of the test case and such interleaving instances are said to be exposed, while the others are referred to as the predicted interleaving instances that are yet to be exposed to exercise more behavior of a concurrent class under test.

Then, a dynamic diversity metric of test cases can be defined as follows.

Definition 5 (*Dynamic diversity metric*). For a test case t and a test case set T such that $t \notin T$, the *dynamic distance* between t and T, denoted as $\nabla(t, T)$, is the number of the interleaving instances that are discovered by t, but have not yet been exposed by any test case in T, i.e., $\nabla(t, T) = |iRoots(t) \setminus IRoots(T)|$.

Technically, $iRoots(t)$ is the set of the interleaving instances discovered by test case t, and can be determined through profiling test case t; $IRoots(T)$ is the set of the interleaving instances exposed by the test cases in set T, and can be incrementally maintained after executing each test case $t' \in T$ actively for the sake of exposing the predicted interleaving instances in $iRoots(t')$. Thus, $\nabla(t, T)$ indicates how many *new* interleaving instances test case t can contribute after the test cases in T have been executed under actively controlled thread schedules.

Both the static and the dynamic diversity metrics quantify the difference between test cases through a concept of *distance* in the test case space. Based on this concept, a diversity-based search can then deliver an *optimal* candidate as the next test case. Intuitively, the more distant (i.e., diverse) a candidate test case is, the more new interleaving instances can be discovered and then exposed. Thus, test cases can be generated in such an adaptive manner for concurrent classes.

5. Adaptive test generation

In this section, we first recall a typical approach of randomly generating candidate test cases [11]. Then, we present three adaptive test generation approaches for concurrent classes, based on the test case diversity metrics presented in Section 4.

5.1. Random candidate test cases

Test cases of a concurrent class C can be randomly generated with the aid of an assistant class AC, which provides actual arguments of certain types required by the methods in class C. To generate a test case t, a sequential prefix p is synthesized by using a randomly selected constructor to define and initialize an object of class C, with the arguments generated randomly or returned from calling a corresponding assistant method in class AC. The object is to be shared between n concurrent suffixes, each composed of a finite sequence of calls to randomly selected methods of the object. The arguments of each method call in a suffix are preferentially the return values from the previous method calls in the suffix or the prefix, if their types match. This is achieved through a backwards search along the suffix and the prefix for the return values of the types explicitly defined in the corresponding method declaration. If such a return value does not exist, a random value is used for a basic type, and a return value from calling a corresponding assistant method in class AC is used for some other type. The arguments passed to the assistant methods are obtained in a similar way. Please refer to [11] for more details about the random test generation approach for Java classes. We implement this approach for C++ classes in this work. As a reminder, unless otherwise specified, the term *test case* refers to a test case of a concurrent class in the rest of this paper. Such randomly generated test cases intend to expose as many interleaving instances as possible between the methods of the concurrent class C.

5.2. Adaptive test generation with the static metric

The principle of FSCS-ART [5] can be directly applied with the static diversity metric. This yields the static diversity-based adaptive test generation approach (S-ATG), presented in Fig. 1, where N is an integer threshold constant, T denotes the set of executed test cases and Z denotes the set of discovered interleaving instances.

S-ATG recursively selects an optimal test case t^* from m random candidates based on the static diversity metric (Line 6). Then, t^* is profiled by running on its own for a couple of times without any control over its thread schedule. The executions of t^* may expose some interleaving instances of the concurrent class, based on which more interleaving instances can be predicted. For example, if $A_X \rightarrow B_X$ is an interleaving instance exposed by profiling t^*, but $B_X \rightarrow A_X$ does not occur

```
1:  $T \leftarrow \emptyset$;
2:  $Z \leftarrow \emptyset$;
3:  $k \leftarrow 0$;
4:  $m \leftarrow N$;
5:  while $k < N$ do
6:      $t^* \leftarrow$ NewTestCase(m);
7:      Profile $t^*$;
8:      $Z' \leftarrow Z \cup iRoots(t^*)$;
9:      Execute $t^*$ for an active test to expose the interleaving
            instances in $Z' \setminus IRoots(T)$;
10:     $T \leftarrow T \cup \{t^*\}$;
11:     if $Z' = Z$ then
12:         $k \leftarrow k + 1$;
13:         $m \leftarrow m + N$;
14:     else
15:         $Z \leftarrow Z'$;
16:         $k \leftarrow 0$;
17:         $m \leftarrow N$;
18:     end if
19: end while
20: function NewTestCase(m)
21:     Generate a candidate set $R$ of $m$ random test cases;
22:     return an optimal test case $t^* \in R$ such that
            for any $t \in R$, $\Delta(t, T) \leq \Delta(t^*, T)$;
23: end function
```

Fig. 1. S-ATG.

in the profiling executions, then both interleaving instances are discovered by t^*, and hence are collected into $iRoots(t^*)$ (Line 8).

Thus, $Z' \setminus IRoots(T)$ consists of all the interleaving instances that have been discovered by test case set $T \cup \{t^*\}$, but have not yet been exposed so far. At Line 9, t^* is executed for an active test, i.e., with thread schedules orchestrated purposely to expose the interleaving instances in $Z' \setminus IRoots(T)$.

If the optimal test case t^* does not introduce any new interleaving instances (i.e., Z already contains all the interleaving instances in $iRoots(t^*)$), S-ATG will increase the size (m) of a candidate set by N in the next round (Line 13). This is to increase the likelihood of discovering more interleaving instances by exploiting a larger candidate set. Once a new interleaving instance is discovered by a test case t^*, m is reset to N to explore a candidate set of initial size in the next round (Line 17). This is to avoid unnecessarily large candidate sets in the search of an optimal test case.

S-ATG terminates when consecutive N executed test cases fail to introduce any new interleaving instances. Note that although the size (m) of a candidate set may be increased or reset back and forth, S-ATG guarantees to terminate eventually when Z remains unchanged for a bounded number (N) of consecutive iterations. This is due to the fact that Z is monotonically non-decreasing after each iteration in S-ATG, and that there are a finite number of interleaving instances between the methods of a concurrent class. Herein, we assume that each method of the concurrent class contains a finite number of instructions.

Instead of executing all the random candidate test cases independently, S-ATG profiles only diverse test cases for active testing. Moreover, all the executed test cases in T are correlated with the memorization $IRoots(T)$ of all the exposed interleaving instances. Thus, the static diversity metric can help eliminate the redundancy in random-based active testing by avoiding scenarios that cannot introduce any new interleaving instances, and by avoiding attempts to expose the same interleaving instances repeatedly.

In the worst case, it would require N iterations to discover a new interleaving instance, where Z remains unchanged for the first $N-1$ iterations, and only in the last (N-th) iteration the new interleaving instance is discovered. Recall that if Z remains unchanged after one iteration, the size of a candidate set will be increased by N in the next iteration. Thus, it would require at most $N + \cdots + N^2 = N^2(N+1)/2$ candidate test cases to discover an interleaving instance. Then, at most $N^2(N+1)(r+1)/2$ random candidate test cases are required to discover r interleaving instances with S-ATG. Note that after the last interleaving instance is exposed, k is reset to 0 (Line 16) and then N more iterations are needed for S-ATG to terminate at the condition $k \geq N$. But during each iteration, only one optimal test case is selected and dispatched for profiling and active testing. Therefore, only $N(r+1)$ test cases are actually compiled, profiled and executed to discover and expose the r interleaving instances.

Let us elaborate in more details on how S-ATG handles the so called "symmetric" test cases. Suppose a test case $t = (p, \{s_i | i \in [1, n]\})$ has been executed for an active test, i.e., $t \in T$. Though very rare, $t^* = (p, \{s_{n-i+1} | i \in [1, n]\}$ may be selected at Line 6, such that t and t^* are symmetric in the sense that t^* can be obtained from t by assigning suffix s_i to the $(n-i+1)$-th thread. Effectively, t^* and t can discover the same interleaving instances. Since t has already been executed, either of the following cases happens for the interleaving instances discovered by t^*:

- All the interleaving instances discovered by t^* have already been exposed successfully, i.e.,

  $iRoots(t^*) \subseteq IRoots(T)$.

  In this case, $Z' = Z$ holds and S-ATG proceeds as usual.
- Some interleaving instance(s) discovered by t^* have already been attempted but not yet exposed, i.e.,

  $iRoots(t^*) \setminus IRoots(T) \neq \emptyset$.

  In this case, t^* is executed at Line 9 for another attempt to expose the interleaving instances in $iRoots(t^*) \setminus IRoots(T)$, and then S-ATG proceeds as usual.

Therefore, S-ATG can well handle such symmetric test cases.

5.3. Adaptive test generation with the dynamic metric

The dynamic diversity metric directly depends on the interleaving instances that a test case can discover. This yields the dynamic diversity-based adaptive test generation approach (D-ATG), which explores the interleaving instances of a concurrent class in a greedy manner. The work-flow of D-ATG is shown in Fig. 2, where N, T and Z have the same definitions as in S-ATG.

Similar to the input-driven active testing approaches we presented in [10], D-ATG maintains a fixed-size (N) candidate set R. All the candidate test cases in R are profiled, but only one optimal test case t^* is selected for active testing in each iteration (Line 17), i.e., an optimal test case that can discover no fewer interleaving instances than any other candidate test case in R (Line 7). Note that profiling a test case, which itself is a multi-threaded program, requires to compile and run the test case non-deterministically. So, the cost of test case profiling is not negligible. Hence, instead of discarding all the candidates in R after each iteration (as in FSCS-ART and S-ATG), D-ATG replaces only the selected test case t^* with a new random candidate t for the next iteration (Line 24).

Condition $\nabla(t^*, T) = 0$ indicates that the dynamic distance between test case t^* and test case set T is 0. This means that all the interleaving instances discovered by t^* have already been exposed by executing the test cases in T (Line 8). Condition $IRoots(T') = IRoots(T)$ with $T' = T \cup \{t^*\}$ indicates that test case t^* cannot expose any but the interleaving instances that have already been exposed by executing the test cases in T (Lines 16 and 15). Due to the various possible way of using a concurrent class in a concurrent program, a single test case t^* satisfying one of these two conditions does not imply a coverage saturation of the interleaving instances.

```
1: $T \leftarrow \emptyset$;
2: $Z \leftarrow \emptyset$;
3: $l \leftarrow 0$;
4: $u \leftarrow 0$;
5: Generate a candidate set $R$ of $N$ random test cases;
6: while $l < N$ and $u < N$ do
7:     Select an optimal test case $t^* \in R$ such that
           for any $t \in R$, $\nabla(t, T) \leq \nabla(t^*, T)$;
8:     if $\nabla(t^*, T) = 0$ then
9:         $l \leftarrow l + 1$;
10:    else
11:        $l \leftarrow 0$;
12:        Profile $t^*$;
13:        $Z' \leftarrow Z \cup iRoots(t^*)$;
14:        Execute $t^*$ for an active test to expose the interleav-
               ing instances in $Z' \backslash IRoots(T)$;
15:        $T' \leftarrow T \cup \{t^*\}$;
16:        if $IRoots(T') = IRoots(T)$ then
17:            $u \leftarrow u + 1$;
18:        else
19:            $u \leftarrow 0$;
20:        end if
21:        $T \leftarrow T'$;
22:    end if
23:    Generate a new random candidate $t$ such that
           $t \notin T \cup R$;
24:    $R \leftarrow (R \backslash \{t^*\}) \cup \{t\}$;
25: end while
```

Fig. 2. D-ATG.

D-ATG is designed with a generalized saturation condition: either $l = N$ or $u = N$. Condition $l = N$ indicates that consecutive N optimal test cases have been selected but cannot discover any new interleaving instances. If this condition holds, there is little chance to discover any more interleaving instances after consecutively selecting N optimal test cases. Condition $u = N$ indicates that consecutive N optimal test cases have been executed but cannot expose any untested interleaving instances. If this condition holds, although there may be still interleaving instances discovered but not yet exposed after consecutively executing N optimal test cases, there is little chance to expose these interleaving instances, or it is highly probable that these interleaving instances are not feasible at all. In summary, if either of the two conditions holds, D-ATG shall terminate with a coverage saturation of the interleaving instances.

D-ATG follows a gradually adaptive strategy such that not all profiled test cases are to be executed for active testing. This reduces the number of active tests without sacrificing the capability of exposing the interleaving instances. In the worst case, it would require active testing of N optimal test cases to expose a feasible interleaving instances, while each of the optimal test cases is obtained by profiling N candidate test cases. Note that the first one initially requires N extra candidate test cases to be profiled. Thus, at most N^2 profiles and N active tests are required to expose a feasible interleaving instance, except for the first feasible interleaving instance which requires N extra profiles. Recall that, after the last feasible interleaving instance is exposed, extra N iterations are mandatory to allow the termination of D-ATG at the condition $l = N$. Then, to expose r feasible interleaving instances with D-ATG, at most $N(Nr + 2)$ candidate test cases are generated and profiled (i.e. $N(Nr + 2)$ profiles), but only Nr optimal test cases are executed for active testing (i.e. Nr active tests). It can be seen that to discover or expose the same number of interleaving instances, the number of candidate test cases generated by S-ATG is $O(N)$ times as much as that by D-ATG; but the number of candidate test cases profiled by D-ATG is $O(N)$ times as much as that by S-ATG, while the numbers of optimal test cases executed for active testing by both approaches are rather comparable. Normally, much less time is consumed to generate a candidate test case randomly than to actually compile and run the candidate test case.

S-ATG and D-ATG are different but orthogonal in their underlying diversity metrics and work-flows. S-ATG selects optimal test cases before profiling, and aims at improving the quality of a random test case to exercise more operation-wise behavior of a concurrent class; while D-ATG selects optimal test cases after profiling but before active testing, and aims at improving the efficiency of active testing in detecting more instruction-wise behavior of the concurrent class. Thus, the two approaches are complementary and hence should be integrated.

5.4. Adaptive test generation with both metrics

An integration of S-ATG and D-ATG yields an adaptive test generation approach based on both the static and the dynamic diversity metrics (SD-ATG). The work-flow of SD-ATG is shown in Fig. 3, where *N, T* and *Z* have the same definitions as in S-ATG and D-ATG.

Similar to D-ATG, SD-ATG also maintains a fixed-size (*N*) candidate set *R*, but each candidate test case in *R*, returned from calling function NEWTESTCASE, is an optimal one among *m* random candidate test cases with respect to the static diversity metric (Lines 8 and 36). The pseudo-code of function NEWTESTCASE is shown in Fig. 1. All the candidate test cases in *R* are profiled, but in each iteration, only one optimal test case is selected from *R* based on the dynamic diversity metric (Line 12), and then executed for active testing (Line 19).

As in S-ATG, *m* is initialized as *N*. If the optimal test case *t** cannot discover any new interleaving instances, *m* is increased by *N* to search for the next test case in a larger candidate set (Line 23); otherwise, if the optimal test case *t** introduces a new interleaving instance, *m* is reset to *N* to search for the next test case in a candidate set of initial size (Line 27). The next test case is then added to the candidate set *R* in replacement of *t** (Line 37).

SD-ATG terminates with a coverage saturation of the interleaving instances, when either of the following three conditions holds:

1. $k = N$, i.e., consecutive *N* optimal test cases cannot discover any new interleaving instances, i.e., $Z' = Z$ with $Z' = Z \cup iRoots(t^*)$ (Lines 21 and 18) for consecutive *N* iterations ;
2. $l = N$, i.e., consecutive *N* optimal test cases cannot discover any new interleaving instance, which is not yet exposed, i.e., $\nabla(t^*, T) = 0$ (Line 13) for consecutive *N* iterations;
3. $u = N$, i.e., consecutive *N* optimal test cases cannot expose any untested interleaving instances, i.e., $IRoots(T') = IRoots(T)$ with $T' = T \cup \{t^*\}$ (Lines 29 and 20) for consecutive *N* iterations.

The first condition is the same as the termination condition in S-ATG, while the last two are the same as the termination conditions in D-ATG. Thus, in SD-ATG, at most *N* profiles are required to deliver an optimal test case for an active test (as in D-ATG), and the *i*-th profiled test case is selected from *iN* random candidate test cases for $1 \leq i \leq N$ (as in S-ATG); in the meanwhile, at most *N* active tests are required to expose a feasible interleaving instance (as in D-ATG). Hence, at most $N(N+\cdots+N^2) = N^3(N+1)/2$ random candidate test cases and N^2 optimal test cases are generated and profiled, respectively, to expose a feasible interleaving instance, except for the first one which requires N^2 extra random candidate test cases to be generated and *N* extra optimal test cases to be profiled. Recall that, as in D-ATG, after the last feasible interleaving instance is exposed, extra *N* iterations are mandatory to allow the termination of D-ATG at the condition $l = N$, which means another N^2 extra random candidate test cases to be generated and another *N* extra optimal test cases to be profiled. Then, to expose *r* feasible

```
1: $T \leftarrow \emptyset$;
2: $Z \leftarrow \emptyset$;
3: $k \leftarrow 0$;
4: $l \leftarrow 0$;
5: $u \leftarrow 0$;
6: $m \leftarrow N$;
7: for each $i \in [1, m]$ do
8:     $t_i \leftarrow$ NewTestCase(m)
9: end for
10: $R \leftarrow \{t_i | i \in [1, m]\}$;
11: while $k < N$ and $l < N$ and $u < N$ do
12:     Select an optimal test case $t^* \in R$ such that
            for any $t \in R$, $\nabla(t, T) \leq \nabla(t^*, T)$;
13:     if $\nabla(t^*, T) = 0$ then
14:         $l \leftarrow l + 1$;
15:     else
16:         $l \leftarrow 0$;
17:         Profile $t^*$;
18:         $Z' \leftarrow Z \cup iRoots(t^*)$;
19:         Execute $t^*$ for an active test to expose the interleav-
                ing instances in $Z \backslash IRoots(T)$;
20:         $T' \leftarrow T \cup \{t^*\}$;
21:         if $Z' = Z$ then
22:             $k \leftarrow k + 1$;
23:             $m \leftarrow m + N$;
24:         else
25:             $Z \leftarrow Z'$;
26:             $k \leftarrow 0$;
27:             $m \leftarrow N$;
28:         end if
29:         if $IRoots(T') = IRoots(T)$ then
30:             $u \leftarrow u + 1$;
31:         else
32:             $u \leftarrow 0$;
33:         end if
34:         $T \leftarrow T'$;
35:     end if
36:     $t \leftarrow$ NewTestCase(m)
37:     $R \leftarrow (R \backslash \{t^*\}) \cup \{t\}$;
38: end while
```

Fig. 3. SD-ATG.

interleaving instances with SD-ATG, at most $N^3(N+1)r/2 + 2N^2$ candidate test cases are generated, but still only $N(Nr+2)$ optimal test cases are profiled, and only Nr optimal test cases are executed for active testing.

The maximal number of test cases generated, profiled and executed for active testing are summarized in Column *Candidate, Profiles* and *Tests*, respectively, in Table 1 for the S-ATG, D-ATG and SD-ATG approaches to expose $r \geq 1$ feasible interleaving instances.

Let ϵ_S, ϵ_D and ϵ_{SD} be the *active execution ratio* of the S-ATG, D-ATG and SD-ATG approach, respectively, between the maximal number of test cases executed for active testing and the maximal number of random candidate test cases generated (i.e., the size of the test case space searched). Intuitively speaking, this ratio indicates the cost-effectiveness of a test generation approach. To expose the same number of interleaving instances, the less the active execution ratio of a test generation approach is, the more cost-effective the approach is. Nevertheless, profiling a test case requires running the test case non-deterministically, which causes extra cost for active testing. Especially in D-ATG and SD-ATG, the number of test case profiled is $O(N)$ times as much as the number of test cases executed for active testing. Let ρ_S, ρ_D and ρ_{SD} be the *profiling ratio* of the S-ATG, D-ATG and SD-ATG approach, respectively, between the maximal number of test case profiled and the maximal number of random candidate test cases generated. It follows directly from Table 1 that, ϵ_S, ϵ_D, ϵ_{SD}, ρ_S, ρ_D and ρ_{SD} are defined as follows:.

$$\begin{aligned}
\epsilon_S &= \frac{N(r+1)}{N^2(N+1)(r+1)/2} = \frac{2}{N(N+1)} \\
\epsilon_D &= \frac{Nr}{N(Nr+2)} = \frac{1}{N+\frac{2}{r}} \\
\epsilon_{SD} &= \frac{Nr}{N^3(N+1)r/2 + 2N^2} = \frac{2}{N^2(N+1) + \frac{4}{r}N} \\
\rho_S &= \frac{N(r+1)}{N^2(N+1)(r+1)/2} = \frac{2}{N(N+1)} \\
\rho_D &= \frac{N(Nr+2)}{N(Nr+2)} = 1 \\
\rho_{SD} &= \frac{N(Nr+2)}{N^3(N+1)r/2 + 2N^2} = \frac{2(N+\frac{2}{r})}{N^2(N+1) + \frac{4}{r}N}
\end{aligned}$$

Interestingly, the upper and lower bounds of the active execution and profiling ratios appear not to correlate with the number of interleaving instances in a concurrent class, as shown in the following formulas:

$$\begin{aligned}
\epsilon_S &= \frac{2}{N(N+1)} \\
\frac{1}{N+2} \leq \epsilon_D &< \frac{1}{N} \\
\frac{2}{N^2(N+1)+4N} \leq \epsilon_{SD} &< \frac{2}{N^2(N+1)} \\
\rho_S &= \frac{2}{N(N+1)} \\
\rho_D &= 1 \\
\frac{2}{N(N+1)} < \rho_{SD} &\leq \frac{2(N+2)}{N^2(N+1)+4N}
\end{aligned}$$

Thus, no matter how many interleaving instances a concurrent class may have, the active execution and profiling ratios of the S-ATG, D-ATG and SD-ATG approaches remain constantly bounded. This shows the characterizations of the scalability of these three approaches in testing complex concurrent classes. Moreover, the parameter N would play an important role in their actual performance in practice.

Table 1
Cost-effectiveness of S-ATG, D-ATG and SD-ATG.

	Candidate	*Profiles*	*Tests*
S-ATG	$N^2(N+1)(r+1)/2$	$N(r+1)$	$N(r+1)$
D-ATG	$N(Nr+2)$	$N(Nr+2)$	Nr
SD-ATG	$N^3(N+1)r/2 + 2N^2$	$N(Nr+2)$	Nr

6. Implementation and experiments

We implement the three adaptive test generation approaches for C/C+ + concurrent data structures, using Maple as the underlying active testing engine. The implementation consists of two parts: a random generator and an adaptive tester. The random generator constructs a random candidate test case in the way as described in Section 4, while the adaptive tester extends Maple with the three adaptive test generation approaches. Thus, our implementation can select random candidate test cases adaptively based on the static diversity metric, the dynamic diversity metric and their combination, and dispatch optimal test cases for active testing until a coverage saturation of interleaving instances is reached for a concurrent data structure under test.

With this implementation, we test 9 open-source concurrent classes (identified as C_1, ..., C_9, respectively), including:

- *LockfreeList* (C_1), which is a typical lock-free concurrent list class based on the *compare-and-swap* (CAS) instruction [32].
- *OptimisticList* (C_2) and *LazyList* (C_3), which are lock-based concurrent list classes using the optimistic synchronization algorithm [33] and the lazy synchronization algorithm [34], respectively.
- *SimpleQueue* (C_4), which is a simple locked-based concurrent queue class.
- *RingQueue* (C_5), which is a concurrent ring queue class with lock-free methods [35].
- *LockfreeQueue* (C_6), which is a lock-free concurrent queue class based on the CAS instruction, and *MSQueue* (C_7), which is a lock-free concurrent queue class using the Michael and Scott algorithm [36].
- *BackoffStack* (C_8), which is a concurrent stack class based on the backoff algorithm [37].
- *LockBasedHashTable* (C_9), which implements a concurrent hash table class with striped locks [38].

The size of each concurrent class is shown in lines of code (Column *LoC*) in Table 2. Our experiments compare the S-ATG, D-ATG, SD-ATG approaches with MRT, which denotes running Maple directly with random test cases until consecutive N random tests cannot discover any new interleaving instances. All the experiments are conducted on a Centos 6.5 server with two Intel Xeon E5-2690 CPUs (32 cores, 2.90 GHz, 24M cache) and 384G memory. Every experiment is repeated 50 times for statistical average. Unless otherwise specified, the saturation threshold N is set to 6. Each test case is equipped with two concurrent suffixes, of which the lengths are not greater than 4. Hence, all the test cases constructed in the experiments are concurrent programs with two threads, while each thread makes a sequence of at most 4 method calls of a shared object.

As a reminder, it was reported that most concurrency bugs can be manifested with no more than two threads [39]. Our recent study [40] also shows that test cases with two threads, one of which even contains only one method call, are sufficient for localizing linearizability bugs of concurrent data structures. Recall that interleaving instances are defined in Section 3 with concurrent instructions from two threads. Therefore, we prefer to using simple test cases for evaluating the performance of our adaptive test generation approaches.

Tables 2, 3, 4 and 5 report the experimental results of the MRT, S-ATG, D-ATG, and SD-ATG approaches, respectively. Column ***iRoots***, *Profiles* and *Tests* report the average numbers (± standard deviation) of interleaving instances exposed, profiles and active tests conducted in testing each concurrent class, respectively. Column *Time* shows the average time consumption in seconds for testing each concurrent class. For simplicity of experiments, we directly measure the total time consumption of consecutively testing a concurrent class 50 times. Hence, each average time consumption reported in Column *Time* is resulted from dividing the corresponding total time consumption by 50. The last rows of these tables report the mean values for each column.

Table 2
Concurrent classes and MRT (*Time* in seconds).

C_i	*LoC*	*iRoots*	*Profiles*	*Tests*	*Time*
1	330	10.4 ± 0.6	16.5 ± 5.0	16.5 ± 5.0	232.4
2	220	28.9 ± 2.7	18.8 ± 4.9	18.8 ± 4.9	143.1
3	200	29.8 ± 2.6	19.0 ± 5.1	19.0 ± 5.1	149.8
4	93	18.5 ± 1.4	15.5 ± 3.0	15.5 ± 3.0	114.4
5	108	25.3 ± 3.9	15.3 ± 4.5	15.3 ± 4.5	109.5
6	173	6.0 ± 0.0	9.8 ± 2.0	9.8 ± 2.0	80.6
7	160	9.3 ± 0.7	8.9 ± 1.8	8.9 ± 1.8	59.9
8	205	4.4 ± 0.5	8.4 ± 1.8	8.4 ± 1.8	51.8
9	534	17.5 ± 1.5	11.4 ± 3.4	11.4 ± 3.4	85.0
Average		16.7	13.7	13.7	114.0

Table 3
S-ATG (*Time* in seconds).

C_i	*iRoots*	*Profiles*	*Tests*	*Time*
1	10.6 ± 0.5	15.7 ± 4.2	15.7 ± 4.2	246.0
2	30.5 ± 2.1	18.7 ± 5.0	18.7 ± 5.0	164.0
3	31.7 ± 2.1	20.0 ± 5.2	20.0 ± 5.2	165.0
4	18.9 ± 1.0	16.0 ± 3.6	16.0 ± 3.6	119.1
5	29.5 ± 4.6	17.9 ± 6.1	17.9 ± 6.1	133.2
6	6.0 ± 0.0	9.2 ± 1.5	9.2 ± 1.5	82.9
7	9.3 ± 0.6	8.7 ± 1.7	8.7 ± 1.7	58.8
8	4.4 ± 0.5	8.0 ± 1.2	8.0 ± 1.2	51.3
9	17.8 ± 1.6	11.9 ± 4.0	11.9 ± 4.0	105.5
Average	17.6	14.0	14.0	125.1
	(5.8%)	(2.2%)	(2.2%)	(9.7%)

Table 4
D-ATG (*Time* in seconds).

C_i	*iRoots*	*Profiles*	*Tests*	*Time*
1	10.5 ± 0.6	13.3 ± 2.2	3.6 ± 1.1	150.2
2	29.3 ± 2.7	18.2 ± 3.8	5.5 ± 1.5	133.9
3	30.3 ± 2.7	19.5 ± 4.8	5.9 ± 1.5	139.0
4	18.0 ± 1.7	13.9 ± 1.9	1.4 ± 0.6	82.6
5	26.8 ± 4.6	17.5 ± 4.5	3.7 ± 1.3	113.8
6	6.0 ± 0.0	12.0 ± 0.1	1.6 ± 0.7	77.4
7	9.4 ± 0.7	12.4 ± 0.7	1.3 ± 0.4	74.2
8	4.7 ± 0.5	12.3 ± 0.5	1.0 ± 0.0	67.1
9	18.0 ± 1.4	13.0 ± 1.8	0.6 ± 0.7	73.4
Average	17.0	14.7	2.7	101.3
	(1.8%)	(7.1%)	(-80.2%)	(-11.2%)

Table 5
SD-ATG (*Time* in seconds).

C_i	*iRoots*	*Profiles*	*Tests*	*Time*
1	10.4 ± 0.6	13.2 ± 2.0	3.7 ± 1.1	159.6
2	30.2 ± 2.3	17.1 ± 3.6	5.4 ± 1.4	151.2
3	31.5 ± 2.2	18.4 ± 4.2	5.8 ± 1.5	148.4
4	18.9 ± 1.0	13.4 ± 1.2	1.3 ± 0.6	87.5
5	29.5 ± 4.3	17.3 ± 4.4	4.1 ± 1.3	126.9
6	6.0 ± 0.0	12.0 ± 0.0	1.3 ± 0.5	81.5
7	9.4 ± 0.5	12.0 ± 0.1	1.2 ± 0.4	77.5
8	4.6 ± 0.5	12.0 ± 0.2	1.0 ± 0.0	68.6
9	18.2 ± 1.4	13.1 ± 2.0	0.7 ± 0.6	85.9
Average	17.6	14.3	2.7	109.7
	(5.8%)	(4.0%)	(-80.3%)	(-3.8%)

6.1. Comparing S-ATG and MRT

In Tables 2 and 3, each number of profiles always equals to the corresponding number of active tests. This is because, in the experiments, all the test cases profiled are dispatched for active testing in both S-ATG and MRT. On average S-ATG can expose 5.8% more interleaving instances than MRT, but at the expenses of 2.2% more profiles and active tests. The experimental results also show that that S-ATG consumes on average 9.7% more time than MRT.

Thus, the static diversity metric can guide Maple to discover and expose more interleaving instances, at the expense of more computation time.

6.2. Comparing D-ATG and MRT

Since in D-ATG, not all profiled candidate test cases are dispatched for active testing, the number of active tests is far less than the corresponding number of profiles in Table 4. On average, although D-ATG conducts 7.1% more profiles than MRT, D-ATG actually executes 80.2% fewer active tests than MRT. This significant reduction in its number of active tests makes D-ATG consume 11.2% less time than MRT. D-ATG can also expose more interleaving instances than MRT. On average D-ATG can expose 1.8% more interleaving instances than MRT.

Thus, the dynamic diversity metric can enhance Maple in two ways: the extra profiles conducted by D-ATG can discover more interleaving instances; while a very significant reduction in the number of active tests by D-ATG also reduces the computation time by a considerable amount. However, D-ATG exposes, on average, less number of interleaving instances than S-ATG.

6.3. *Comparing SD-ATG and MRT*

Through combing the static and the dynamic diversity metrics, SD-ATG achieves a more well-balanced performance for testing of concurrent data structures, with respect to the following two aspects: on average,

- SD-ATG can expose 5.8% more interleaving instances than MRT, at the expense of 4.0% more profiles.
- SD-ATG consumes 3.8% less time than MRT, which is mainly due to a very significant (80.3%) reduction in the number of active tests conducted by SD-ATG.

It can be seen that SD-ATG well takes the advantages of the static and dynamic diversity metrics, while their disadvantages are offset through the integration of both metrics in the SD-ATG framework. Compared to S-ATG, SD-ATG can expose more interleaving instances without requiring more computation time. Compared to D-ATG, SD-ATG can expose more interleaving instances.

6.4. *Efficiency in exposing interleaving instances*

As observed in the above experiments, S-ATG, D-ATG and SD-ATG consistently outperform MRT in the number of interleaving instances exposed. We further analyze the average performance of the four approaches in exposing one interleaving instance, as shown in Table 6. We put the average statistics reported in the last rows of Tables 2, 3, 4 and 5 together in the columns 2–4 of Table 6.

The interleaving instances of a concurrent data structure are exposed either by directly profiling a test case, or by executing a test case for an active test. Column 5 of Table 6 shows the average number of profiles and active tests conducted by each approach to expose an interleaving instance, and the last column reports the average time consumption for each approach to expose an interleaving instance.

On one hand, by integrating with the static diversity metric, S-ATG can expose more interleaving instances than MRT, while SD-ATG can expose more interleaving instances than D-ATG. But, in both cases, the static diversity metric may introduce a slightly extra cost for exposing an interleaving instance.

On the other hand, by integrating with the dynamic diversity metric, D-ATG can expose more interleaving instances than MRT, but SD-ATG exposes roughly the same number of interleaving instances as S-ATG does. But, in both cases, the dynamic diversity metric can help reduce the average number of profiles and active tests required for exposing an interleaving instance by about $(1.6 - 1.0)/1.6 \approx 38\%$, and reduce the average time consumption for exposing an interleaving instance by about $(6.8 - 6.0)/6.8 \approx (7.1 - 6.2)/7.1 \approx 12\%$.

Table 6
Performance per interleaving instance.

	Average	*Average*	*Average*	*Average*	*Profiles + Tests*	*Time*
	iRoots	*Profiles*	*Tests*	*Time*	*per iRoot*	*per iRoot*
MRT	16.7	13.7	13.7	114.0	1.6	6.8
S-ATG	17.6	14.0	14.0	125.1	1.6	7.1
D-ATG	17.0	14.7	2.7	101.3	1.0	6.0
SD-ATG	17.6	14.3	2.7	109.7	1.0	6.2

6.5. *Detecting the first concurrency bug*

The purpose of exposing the interleaving instances of a concurrent data structure is to exercise its behavior under all the possible thread schedules. In doing so, the faulty behavior can be identified, which indicates the existence of a bug in the concurrent data structure. In this subsection, we implant a concurrency bug to each of the above concurrent classes, and then evaluate the efficiency of the MRT, S-ATG, D-ATG, SD-ATG approaches in detecting the first concurrency bug. The implanted bugs are to be detected through assertion violations in the mutants.

The corresponding experimental results are reported in Table 7, where Columns *Profiles, Tests, Time* present the same types of the statistics as reported in previous tables. In the column of *Mutant*, C_{ij} is the j-th mutant of class C_i.

- C_{11} is obtained by replacing an atomic CAS instruction in the *insert* function of class *LockfreeList* (C_1), with a non-atomic CAS function.
- C_{21} is obtained by commenting out the lock and unlock instructions in the *insert* function of class *OptimisticList* (C_2). C_{22} is another mutant of C_2 by commenting out only the lock instructions in the same function.
- C_{31} and C_{32} are obtained from *LazyList* (C_3) in the same way as C_{21} and C_{22} are obtained from C_2, respectively.
- C_{41} and C_{42} are obtained by commenting out the lock and unlock instructions in the *enqueue* and *dequeue* function of class *SimpleQueue* (C_4), respectively. C_{43} and C_{44} are obtained by commenting out only the lock instructions in the *enqueue* and *dequeue* function of class C_4, respectively.
- C_{91} is obtained from class *LockBasedHashTable* (C_9) by commenting out only the unlock instructions in the *insert* function.

It can be seen that to detect the first concurrency bug, on average, S-ATG needs 11.5% fewer profiles and active tests than MRT; while D-ATG needs 95.6% more profiles but 57.9% fewer active tests than MRT. Through integrating with S-ATG, SD-ATG outperforms D-ATG as it needs 82% more profiles but 62.9% fewer active tests than MRT.

Similar observations can be made on the computation time required by each approach. To detect the first concurrency bug, on average, S-ATG consumes 11.8% less time than MRT, while D-ATG and SD-ATG consume 18.% and 31.2% more time than MRT.

The experimental results reflect the inherent complexity in computing the dynamic diversity metric, which requires profiling candidate test cases. It may take even more time for SD-ATG to find an optimal test case because it has to compute both the static and the dynamic diversity metrics. Hence, if a concurrency bug is not very difficult to detect, the dynamic diversity metric may cause extra computation burden for detecting the bug. Comparatively, the static diversity metric can help improve the quality of random test cases in detecting a concurrency bug, especially the first bug. This actually coincides with the efficiency improvement observed on the adaptive random testing approaches [6].

However, most concurrency bugs can only be detected under subtle and specific interleaving instances. As discussed above, the dynamic diversity metric can help reach the same level of interleaving coverage more efficiently than the static diversity metric. Thus, with the aid of the dynamic diversity metric, the subtle interleaving instances can be discovered and then exposed more efficiently, for detecting a heisenbug [17].

Furthermore, the above experiments also indicate that the dynamic diversity metric can reduce the number of active tests by introducing

Table 7
Detecting the first concurrency bug (*Time* in seconds).

Mutant	MRT			S-ATG			D-ATG			SD-ATG		
	Profiles	*Tests*	*Time*	*Profiles*	*Tests*	*Time*	*Profiles*	*Tests*	*Time*	*Profiles*	*Tests*	*Time*
C_{11}	7.7 ± 5.7	7.7 ± 5.7	108.0	5.7 ± 3.5	5.7 ± 3.5	98.4	8.0 ± 2.5	2.2 ± 1.5	98.8	7.8 ± 2.7	2.2 ± 1.6	139.5
C_{21}	3.2 ± 3.1	3.2 ± 3.1	50.1	2.9 ± 1.9	2.9 ± 1.9	49.2	7.4 ± 3.1	1.7 ± 1.5	51.8	6.4 ± 1.1	1.4 ± 1.1	54.1
C_{22}	3.8 ± 3.2	3.8 ± 3.2	48.4	3.4 ± 2.5	3.4 ± 2.5	32.6	7.3 ± 2.7	1.8 ± 1.5	44.5	6.4 ± 0.7	1.3 ± 0.6	45.3
C_{31}	4.6 ± 4.3	4.6 ± 4.3	50.5	4.6 ± 3.5	4.6 ± 3.5	35.1	7.6 ± 2.6	2.0 ± 1.4	54.5	6.9 ± 1.5	1.7 ± 1.2	55.0
C_{32}	4.5 ± 5.1	4.5 ± 5.1	46.4	4.3 ± 3.4	4.3 ± 3.4	40.3	8.0 ± 3.1	2.1 ± 1.8	54.5	6.9 ± 1.7	1.7 ± 1.2	58.5
C_{41}	1.9 ± 1.3	1.9 ± 1.3	14.0	1.3 ± 0.6	1.3 ± 0.6	13.1	6.2 ± 1.0	1.1 ± 0.3	26.7	6.0 ± 0.0	1.0 ± 0.0	24.2
C_{42}	1.8 ± 1.9	1.8 ± 1.9	10.6	1.9 ± 1.1	1.9 ± 1.1	13.8	6.4 ± 1.4	1.1 ± 0.3	35.0	6.4 ± 1.5	1.0 ± 0.1	36.5
C_{43}	1.7 ± 0.9	1.7 ± 0.9	13.3	1.5 ± 0.8	1.5 ± 0.8	14.7	6.1 ± 0.8	1.1 ± 0.2	27.6	6.0 ± 0.0	1.0 ± 0.0	26.8
C_{44}	1.7 ± 1.1	1.7 ± 1.1	19.7	1.7 ± 1.3	1.7 ± 1.3	16.1	6.8 ± 2.2	1.1 ± 0.3	30.1	6.4 ± 1.7	1.0 ± 0.2	35.4
C_{91}	6.3 ± 3.6	6.3 ± 3.6	50.7	5.7 ± 4.5	5.7 ± 4.5	49.9	9.2 ± 3.5	1.5 ± 0.6	64.6	8.7 ± 3.3	1.4 ± 0.6	65.0
Average	3.7	3.7	41.2	3.3 (-11.5%)	3.3 (-11.5%)	36.3 (-11.8%)	7.3 (95.6%)	1.6 (-57.9%)	48.8 (18.6%)	6.8 (82.0%)	1.4 (-62.9%)	54.0 (31.2%)

moderately more profiles. Note that profiling a test case does not need any external intervention in running the test case. Thus, the dynamic diversity metric also saves the test efforts spent for actively controlling the execution of a test case, which is a quite difficult task for a multi-threaded test case.

6.6. Different values of N

In the above experiments, the saturation threshold *N* is fixed at 6. We further evaluate the impact of this threshold value on the number of exposed interleaving instances and the computation time, for the MRT, S-ATG, D-ATG and SD-ATG approaches. The results of testing *OptimisticList* with various values of *N* are shown in Table 8.

It can be seen that on average, steadily more interleaving instances can be exposed by increasing the threshold, as shown by the solid curves in Fig. 4. The standard deviation of the number of exposed interleaving instances is monotonically decreasing. This suggests that, the larger the threshold is, the more stable the number of exposed interleaving instances is. However, the computation time of each approach increases dramatically with the increasing threshold, as shown by the dashed curves in Fig. 5. It is still an open question whether there exists an optimal value for the threshold N.

Recall that there are a finite number of interleaving instances in a concurrent class. Hence, as the threshold *N* increases, the number of the interleaving instances exposed by each approach tends to converge eventually. For S-ATG and SD-ATG, *N* also defines the size of a candidate set, from which an optimal test case is selected. Thus, the experimental results also support the intuition the larger a candidate set is, the more interleaving instances S-ATG and SD-ATG can discover and expose.

7. Limitations

This section discusses the limitations of this work from the perspective of testing concurrent data structures.

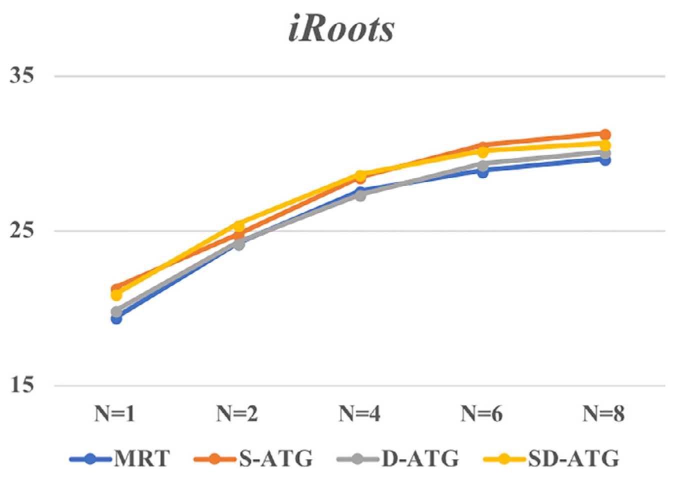

Fig. 4. Number of interleaving instances exposed under different values of *N*.

Adaptive Random Testing. This work is motivated by the adaptive random testing approach [6], which has illustrated the fundamental role of diversity in test case selection strategies. We envisage that this principle can fit in the context of testing a concurrent data structure, i.e., intuitively, the more diverse the test cases executed for active testing are, the more interleaving instances and hence the more behavior of the concurrent data structure can be examined. The D-ATG and SD-ATG approaches presented in this paper do not quite follow the typical work-flow of adaptive random testing, because all but the first executed test cases are not necessarily selected from fresh random candidates, while all executed test cases are in adaptive random testing. However, our experiments show that, compared to the original active testing approach with naively random test cases, the S-ATG, D-ATG and SD-ATG approaches all efficiently achieve a higher coverage of the interleaving instances for testing the concurrent data

Table 8
Different values of *N*.

N	MRT		S-ATG		D-ATG		SD-ATG	
	iRoots	*Time*	*iRoots*	*Time*	*iRoots*	*Time*	*iRoots*	*Time*
1	19.4 ± 5.6	41.6	21.3 ± 5.0	47.0	19.9 ± 6.1	42.7	20.9 ± 5.5	44.3
2	24.2 ± 4.9	72.0	24.8 ± 4.6	72.3	24.2 ± 3.6	67.2	25.4 ± 4.7	71.5
4	27.6 ± 2.8	109.9	28.5 ± 2.4	119.3	27.3 ± 2.8	105.8	28.7 ± 2.3	114.6
6	28.9 ± 2.7	143.1	30.5 ± 2.1	164.0	29.3 ± 2.7	133.9	30.2 ± 2.3	151.2
8	29.7 ± 2.0	182.3	31.3 ± 2.0	252.8	30.1 ± 1.9	208.9	30.7 ± 1.8	222.6

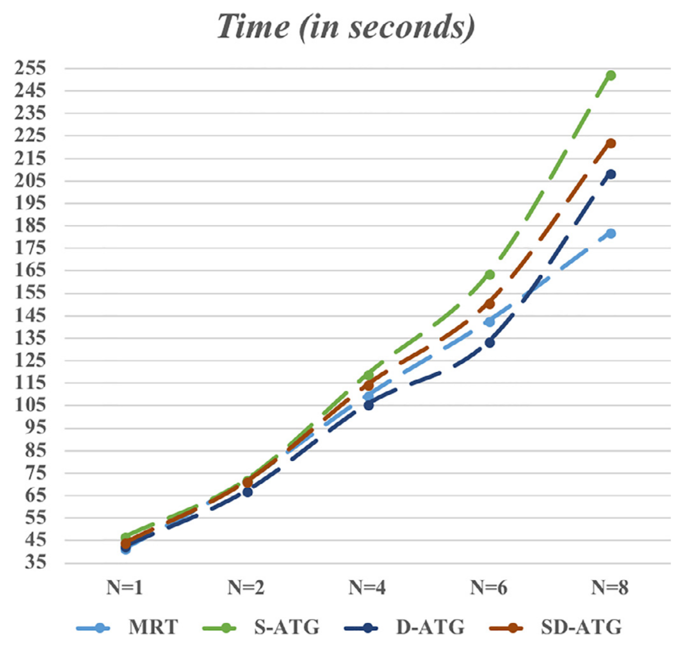

Fig. 5. Time consumption under different values of *N*.

structure. This is exactly the effect expected for diverse test cases.

Black-Box Testing. In the adaptive test generation approaches, all candidate test cases are constructed randomly without referring to the internal implementation of a concurrent data structure under test. However, in D-ATG and S-ATG, not all candidates, but only the optimal ones with respect to the dynamic diversity metric, are actually executed for active testing. The dynamic diversity metric takes into account the run-time behavior of a candidate test case (or precisely, the concurrent data structure). From this perspective, S-ATG can be regarded as a black-box testing approach, because it selects the optimal test cases based solely on the static diversity metric, without any reference to the concrete implementation of the concurrent data structure.

Linearizability. The testing framework presented in this paper mainly addresses the test generation and execution problems in testing concurrent data structures, and pays less attention to its test oracle problem [41]. A test oracle is a mechanism to determine whether a test case is executed correctly by a software under test. Linearizability [42] is a widely accepted correctness criterion of concurrent data structures. Intuitively, a concurrent execution of a shared object is linearizable if each operation of the object appears to take effect instantaneously at some point between the invocation and the response of the operation, and the execution can be serialized as a sequence of operations that is consistent with the sequential specification of the object. In Section 6.5, the implanted concurrency bugs are detected through explicit assertions, which are developed manually based on the sequential specifications of concurrent data structures. However, in general, testing a concurrent execution for linearizability is NP-complete [43].

8. Conclusions and future work

In this paper, we have proposed the static and dynamic diversity metrics for test cases of concurrent data structures, and the three adaptive approaches of generating multi-threaded programs for testing concurrent data structures, namely, S-ATG, D-ATG and SD-ATG. These approaches execute only optimal test cases which are selected according to their relevant diversity metrics for active testing. The experimental results from 9 open-source concurrent classes demonstrate the effectiveness of the static and dynamic diversity metrics in guiding the selection of diverse test cases. The static diversity metric can help expose more interleaving instances, but at the expense of more computation time; while the dynamic diversity metric can dramatically reduce the number of active tests required, with a high performance in exposing interleaving instances.

This work further confirms that the principle of test case diversity can improve the efficiency in testing concurrent data structures, even though their test cases are multi-threaded programs. Test case diversity metrics play a vital role in the performance of our adaptive test generation approaches.

As for future work, we would like to study the interleaving instances between more than two threads, which inherently require concurrent test cases with more than two threads. Furthermore, it is natural but challenging to seek a diversity metric that is not only based on the semantics of test cases, but also is able to enjoy a low computational complexity. Such a diversity metric would support a cost-effective way to detect heisenbugs in concurrent data structures. The idea of adaptive test generation can also be adapted for testing massively parallel software with loosely coupled threads/processes. Testing concurrent software has not yet been fully explored. This work has given some insights on the potential of developing diversity metrics based on the correctness criteria of a concurrent software under test, such as, linearizability. In this way, the test case problem and the oracle problem of testing concurrent software can be studied in a unified framework.

Acknowledgments

We thank the anonymous referees for their valuable comments. This work is partially supported by 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, the CAS-INRIA Joint Research Program under the Grant No. GJHZ1844 and the National Natural Science Foundation of China under the Grant No. 61100069.

References

[1] J. Yu, S. Narayanasamy, C. Pereira, G. Pokam, Maple: a coverage-driven testing tool for multithreaded programs, Proceedings of ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12), (2012), pp. 485–502.
[2] T.Y. Chen, F.-C. Kuo, D. Towey, Z. Zhou, A revisit of three studies related to random testing, Sci. China Inf. Sci. 58 (5) (2015) 1–9.
[3] T.Y. Chen, Y.T. Yu, On the relationship between partition and random testing, IEEE Trans. Software Eng. 20 (12) (1994) 977–980.
[4] P. Ammann, J. Offutt, Introduction to Software Testing, Cambridge University Press, 2008.
[5] T.Y. Chen, H. Leung, I.K. Mak, Adaptive random testing, Proceedings of the 9th Asian Computing Science Conference. Advances in Computer Science - ASIAN 2004, Higher-Level Decision Making. LNCS 3321, (2005), pp. 320–329.
[6] T.Y. Chen, F.-C. Kuo, R.G. Merkel, T.H. Tse, Adaptive random testing: the art of test case diversity, J. Syst. Software 83 (1) (2010) 60–66.
[7] A.C. Barus, T.Y. Chen, F.-C. Kuo, H. Liu, R. Merkel, G. Rothermel, A cost-effective random testing method for programs with non-numeric inputs, IEEE Trans. Comput. 65 (12) (2016) 3509–3523.
[8] T.Y. Chen, R. Merkel, An upper bound on software testing effectiveness, ACM Trans. Software Eng. Methodol. (TOSEM) 17 (3) (2008) 16:1–16:27.
[9] J. Chen, F.-C. Kuo, T.Y. Chen, D. Towey, C. Su, R. Huang, A similarity metric for the inputs of OO programs and its application in adaptive random testing, IEEE Trans. Reliab. 66 (2) (2016) 373–402.
[10] H. Yue, P. Wu, T.Y. Chen, Y. Lv, Input-driven active testing of multi-threaded programs, Proceedings of 2015 Asia-Pacific Software Engineering Conference (APSEC '15), (2015), pp. 246–253.
[11] M. Pradel, T.R. Gross, Fully automatic and precise detection of thread safety violations, Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12), (2012), pp. 521–530.
[12] A. Choudhary, S. Lu, M. Pradel, Efficient detection of thread safety violations via coverage-guided generation of concurrent tests, Proceedings of the 39th International Conference on Software Engineering (ICSE '17), (2017), pp. 266–277.
[13] M. Popovic, I. Basicevic, Test case generation for the task tree type of architecture, Inf. Softw. Technol. 52 (6) (2010) 697–706.
[14] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, D. Holmes, Java Concurrency in Practice, Addison-Wesley Professional, 2005.
[15] P. Godefroid, Model checking for programming languages using Verisoft,

Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL '97), (1997), pp. 174–186.

[16] M. Musuvathi, S. Qadeer, CHESS: systematic stress testing of concurrent software, Logic-Based Program Synthesis and Transformation, LNCS 4407, (2007), pp. 15–16.

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

[18] K. Sen, Race directed random testing of concurrent programs, Proceedings of the 29th ACM International Conference on Programming Language Design and Implementation (PLDI '08), (2008), pp. 11–21.

[19] W. Zhang, C. Sun, S. Lu, Conmem: Detecting severe concurrency bugs through an effect-oriented approach, Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV), (2010), pp. 179–192.

[20] B. Křena, Z. Letko, T. Vojnar, Coverage metrics for saturation-based and search-based testing of concurrent software, Proceedings of the 2nd International Conference on Runtime Verification (RV '11), (2012), pp. 177–192.

[21] J. Fiedor, V. Hrubá, B. Křena, Z. Letko, S. Ur, T. Vojnar, Advances in noise-based testing of concurrent software, Software Test. Verif. Reliab. 25 (3) (2015) 272–309.

[22] S. Hong, J. Ahn, S. Park, M. Kim, M.J. Harrold, Testing concurrent programs to achieve high synchronization coverage, Proceedings of 2012 International Symposium on Software Testing and Analysis (ISSTA '12), (2012), pp. 210–220.

[23] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, S. Ur, Multithreaded Java program test generation, IBM Syst. J. 41 (1) (2002) 111–125.

[24] S. Burckhardt, P. Kothari, M. Musuvathi, S. Nagarakatte, A randomized scheduler with probabilistic guarantees of finding bugs, Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV), (2010), pp. 167–178.

[25] P. Godefroid, J. Van Leeuwen, J. Hartmanis, G. Goos, P. Wolper, Partial-order methods for the verification of concurrent systems: An approach to the state-explosion problem, LNCS 1032, Springer Heidelberg, 1996.

[26] C. Flanagan, P. Godefroid, Dynamic partial-order reduction for model checking software, Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05), (2005), pp. 110–121.

[27] M.L. Scott, Shared-Memory Synchronization, Morgan & Claypool Publishers, 2013.

[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, K. Hazelwood, Pin: Building customized program analysis tools with dynamic instrumentation, Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05), (2005), pp. 190–200.

[29] G. Navarro, A guided tour to approximate string matching, ACM Comput. Surv. (CSUR) 33 (1) (2001) 31–88.

[30] V.I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals, Soviet Physics Doklady 10 (8) (1966) 707–710.

[31] R.A. Wagner, M.J. Fischer, The string-to-string correction problem, J. ACM (JACM) 21 (1) (1974) 168–173.

[32] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann, 2011.

[33] M. Herlihy, Y. Lev, V. Luchangco, N. Shavit, A simple optimistic skiplist algorithm, Proceedings of the 14th International Conference on Structural Information and Communication Complexity (SIROCCO '07), (2007), pp. 124–138.

[34] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W.N. Scherer III, N. Shavit, A lazy concurrent list-based set algorithm, Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS '05), (2005), pp. 3–16.

[35] A fast no_lock RingQueue with multi threads.

[36] M.M. Michael, M.L. Scott, Simple, fast, and practical non-blocking and blocking concurrent queue algorithms, Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96), (1996), pp. 267–275.

[37] D. Hendler, N. Shavit, L. Yerushalmi, A scalable lock-free stack algorithm, Proceedings of the 16th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '04), (2004), pp. 206–215.

[38] Concurrent hashmap implementation with striped lock.

[39] S. Lu, S. Park, E. Seo, Y. Zhou, Learning from mistakes: a comprehensive study on real world concurrency bug characteristics, Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII), (2008), pp. 329–339.

[40] Z. Zhang, P. Wu, Y. Zhang, Localization of linearizability faults on the coarse-grained level, Int. J. Software Eng. Knowl. Eng. 27 (09n10) (2017) 1483–1505.

[41] E.T. Barr, M. Harman, P. McMinn, M. Shahbaz, S. Yoo, The Oracle problem in software testing: a survey, IEEE Trans. Software Eng. 41 (5) (2015) 507–525.

[42] M.P. Herlihy, J.M. Wing, Linearizability: a correctness condition for concurrent objects, ACM Trans. Program. Lang. Syst. (TOPLAS) 12 (3) (1990) 463–492.

[43] P.B. Gibbons, E. Korach, Testing shared memories, SIAM J. Comput. 26 (4) (1997) 1208–1244.