Best Paper Award at ICST '13

CHECK-THEN-ACT Misuse of Java Concurrent Collections

Yu Lin and Danny Dig
University of Illinois at Urbana-Champaign
{*yulin2, dig*}*@illinois.edu*

***Abstract*—Concurrent collections provide thread-safe, highly-scalable operations, and are widely used in practice. However, programmers can misuse these concurrent collections when composing two operations where a check on the collection (such as non-emptiness) precedes an action (such as removing an entry). Unless the whole composition is atomic, the program contains an atomicity violation bug.**

In this paper we present the first empirical study of CHECK-THEN-ACT idioms of Java concurrent collections in a large corpus of open-source applications. We catalog nine commonly misused CHECK-THEN-ACT idioms and show the correct usage. We quantitatively and qualitatively analyze 28 widely-used open source Java projects that use Java concurrency collections – comprising 6.4M lines of code. We classify the commonly used idioms, the ones that are the most error-prone, and the evolution of the programs with respect to misused idioms. We implemented a tool, CTADETECTOR, to detect and correct misused CHECK-THEN-ACT idioms. Using CTADETECTOR we found 282 buggy instances. We reported 155 to the developers, who examined 90 of them. The developers confirmed 60 as new bugs and accepted our patch. This shows that CHECK-THEN-ACT idioms are commonly misused in practice, and correcting them is important.

I. INTRODUCTION

The hardware industry keeps up with Moore's law by resorting to multicore processing. Nowadays multicores are everywhere: in smart phones, tablets, laptops, and desktop computers. In the multicore era, the software industry can benefit from hardware improvements if they leverage concurrent programming. However, writing concurrent programs is hard: the programmer has to balance two conflicting forces, thread-safety and performance.

The industry trend is to convert the hard problem of using concurrency into the easier problem of using a concurrent library. For example, Microsoft provides Task Parallel Library (TPL) [1] and Collections.Concurrent (CC [2]), Intel provides the Threading Building Blocks (TBB) [3], and the Java community uses the `java.util.concurrent` (`j.u.c.`) [4] library.

According to previous empirical studies of concurrent library usage [5]–[8], concurrent collections are one of the most widely-used features. Concurrent collections (e.g., `ConcurrentHashMap` from `j.u.c.`) contain thread-safe, scalable data structures. Their individual operations are thread-safe. For example, several threads can safely `put` into the same `ConcurrentHashMap` in parallel.

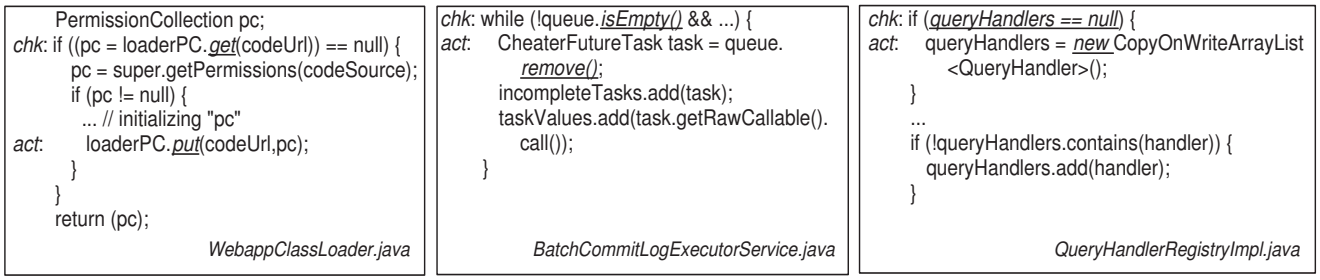
However, concurrent collections can be easily misused. Often programmers combine several operations to express higher-level semantics such as CHECK-THEN-ACT [7] idioms. In this idiom, the code first checks a condition, and then acts based on the result of the condition.

Figure 1 shows three real-world examples of CHECK-THEN-ACT idioms. The labels `chk` and `act` mark the check and act operations, respectively. In Fig. 1(a) the code checks whether a `ConcurrentHashMap loaderPC` contains a specific key and if it does not, the code creates a new value and puts it into the map. In Fig. 1(b) the code checks whether the queue is empty, and if not, it removes elements from the queue. Figure 1(c) shows a classic lazy-initialization: the code checks whether a list reference is `null`, and if so, it creates a new list and adds elements into the list.

All three examples lead to bugs when they are executed under concurrent threads, say T_1 and T_2. In Fig. 1(a), suppose that both T_1 and T_2 execute statement `chk` and find that the map does not contain the key. Thus, they both calculate the value and put it into the map. Whoever is the last one will overwrite the value put by the other thread. This breaks the put-if-absent semantics of the original code. In Fig. 1(b), if T_2 removes the last element from the queue while T_1 is between `chk` and `act`, the element retrieved by T_1 will be null, which will lead to a `NullPointerException` in the fifth line of code. In Fig. 1(c) suppose that both threads find the list field is `null` and initialize it. In this case one initialization will override the other. The elements added by one thread will be lost. We found and reported all three bugs to the developers, who confirmed them as new bugs and applied our patch.

Notice that these are all examples of atomicity violation bugs: an operation executed by a thread T_2 between the T_1 thread's execution of `chk` and `act` statements might make T_1 act based on a stale condition. This can result in corrupted data structures, null pointer exceptions, and semantic errors (e.g., overwrite). Such errors can occur even if the programmers use concurrent libraries, as shown in our three examples. We call the above errors *semantic bugs*.

In addition to semantic bugs, programmers can also introduce *performance bugs* when using CHECK-THEN-ACT. A performance bug is an over-synchronized CHECK-THEN-ACT idiom that harms the performance. For the example in Fig. 1(a), suppose that the programmer used a lock to make the CHECK-THEN-ACT idiom atomic. However, this



(a) A usage of ConcurrentHashMap in *GlassFish* (b) A usage of BlockingQueue in *Cassandra* (c) A usage of CopyOnWriteArrayList in *CXF*

Figure 1. Three instances of misused CHECK-THEN-ACT idioms of concurrent collections used in real-world applications.

reduces the scalability of the application, because the same lock is used to protect all other access to the `loaderPC` map. While this correctly prevents overlapping read and write concurrent accesses, it also prevents concurrent read access. In an application with predominantly read accesses, the lock-based synchronization dramatically reduces the performance. A much better approach is to use the compound update APIs provided in the concurrent collections. In this example, we can change the code to use `ConcurrentHashMap.putIfAbsent`.

In this paper we present the first empirical study that answers in-depth questions about the usage of CHECK-THEN-ACT idioms on a large scale. Our corpus contains 28 widely-used open-source Java projects that use concurrent collections. These projects comprise 6.4M non-blank, non-comment source lines of code (SLOC). We implemented a tool, CTADETECTOR, which uses a static analysis approach to detect instances of misused idioms and a semi-automated transformation approach to correct them. Using this data and our tool, we answer four research questions:

RQ1: What are the commonly used CHECK-THEN-ACT idioms in real-world programs? We found that in each category of correctly used and misused idioms, there is one idiom that clearly dominates the others.

RQ2: Which idioms are the most error-prone? We found one single idiom, `put-if-absent`, for which the number of misused instances is larger than the number of correctly used instances.

RQ3: Do misused idioms result in real bugs? Are our patches accepted by developers? We found 282 misused instances (217 semantic and 65 performance-related). So far we reported 155 bugs to developers, and they examined 90 of them. The developers confirmed 60 of the examined buggy instances as new bugs. For these confirmed bugs, the developers accepted the patches generated by CTADETECTOR. The developers claim that the remaining instances do not lead to real bugs because the buggy interleaving can not not occur in practice, or the programs are resilient to such bugs.

RQ4: What is the evolution of programs w.r.t. CHECK-THEN-ACT idioms? We found that across three major versions between 2007 to 2012, the number of both correct and incorrect usages increase. However, in the later versions, the percentage of incorrect usage decreases.

There are several implications of our findings. Programmers learn a new programming construct through both positive and negative examples. Our catalog of idioms teaches them how to use CHECK-THEN-ACT idioms correctly. Along with the hundreds of instances of idioms, it provides a tremendous educational resource.

Second, library designers can use our findings to make the APIs more robust or provide better documentation. Third, the testing community can focus its effort to find CHECK-THEN-ACT bugs in concurrent programs.

This paper makes the following contributions:

1. Catalog of idioms: To the best of our knowledge, we are the first to catalog the incorrect usage of CHECK-THEN-ACT idioms of concurrent collections.

2. Analysis of instances: By mining 28 projects, we uncover 282 misused and 545 correctly used instances of the idioms. Using this data, we answer questions about popularity, error-proneness, and evolution of idioms. This data lead to the discovery of 60 new bugs, confirmed by the developers.

3. Tool for detection and correction: We implemented a pattern-based static analysis tool, CTADETECTOR, to detect misused CHECK-THEN-ACT idioms. To correct the misused idioms, our tool uses an *interactive* program-transformation approach.

All our data, bug reports, and the tool are available at: http://mir.cs.illinois.edu/~yulin2/CTADetector/

II. CATALOG OF IDIOMS

By default, most of the Java collection classes are not thread-safe. Therefore, the `j.u.c.` package introduces several *thread-safe* concurrent collections, e.g., `ConcurrentHashMap`, `BlockingQueue`, and `CopyOnWriteArrayList`. Before the introduction of `j.u.c.`, a programmer could create a thread-safe `HashMap` using a synchronized wrapper (e.g., `Collections.synchronizedMap(aMap)`). The synchronized `HashMap` achieves its thread-safety by protecting all accesses to the map with a common lock. This results in poor scalability when multiple threads try to access

```
(a)Value v = map.get(key);
   if(v == null){
      v = calc();
      map.put(key, v);
      ...
   }

(b)if(map.get(key) == null){
      v = calc();
      map.put(key, v);
      ...
   }

(c)Value v = map.get(key);
   if(v != null){
      ...
      return;
   }
   v = calc();
   map.put(key, v);
   ...

(d)if(map.get(key) != null){
      ...
      return;
   }
   v = calc();
   map.put(key, v);
   ...

(e)if(!map.containsKey(key)){
      v = calc();
      map.put(key, v);
      ...
   }

(f)if(map.containsKey(key)){
      ...
      return;
   }
   v = calc();
   map.put(key, v);
   ...
```

Figure 2. `Put-if-absent` idiom and its variations for `ConcurrentHashMap`.

different parts of the map simultaneously, since they contend for the same lock.

The concurrent collections include the API methods offered by their corresponding non-thread safe counterparts. In addition, they contain new APIs that encapsulate compound update operations, and execute atomically, without resorting to one common lock. Using the concurrent collections over the synchronized collections offers dramatic scalability improvements [7]. However, it is still possible to introduce bugs when using concurrent collections.

Terminology: In this paper we use the term *idiom* to refer to a recurring programming construct that developers use when working with concurrent collections. Like design patterns [9], the idioms abstract away the details from code. We call an *instance of an idiom* a concrete incarnation of the idiom in real code.

The widely-used CHECK-THEN-ACT idiom can be expressed as specific idioms for specific collections (e.g., `put-if-absent` for `ConcurrentHashMap`). An idiom, can also have syntactical variations (e.g., by using different API methods), even for the same collection.

We classify an idiom as *misused* when it can result in a non-atomic execution of the check and act operations (semantic problems) or it is over-synchronized (performance problems). In some cases, this can manifest as a disuse of the atomic library APIs or an erroneous use, in others as over-zealous synchronization. We simply call all of them a misuse of the concurrent collection API.

In this section we present misused CHECK-THEN-ACT idioms for individual collections and explain how these idioms can lead to semantic or performance bugs. We conclude the section by summarizing the common traits of these idioms and correction strategies.

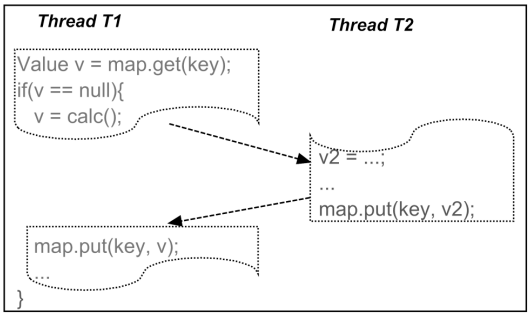


Figure 3. An example of buggy interleaving for Fig. 2(a)

A. *Misused* CHECK-THEN-ACT *in* `ConcurrentHashMap`

`ConcurrentHashMap` is a thread-safe implementation of `HashMap`. In addition, it contains three new APIs: `putIfAbsent(key, value)`, `replace(key, oldValue, newValue)`, and a conditional `remove(key, value)`. For example, `putIfAbsent` (1) checks whether the map contains a given key, and (2) if absent, inserts the $\langle key, value \rangle$ entry. This is a classic example of a CHECK-THEN-ACT idiom. The library guarantees that these two steps are done atomically.

Next, we present the misused CHECK-THEN-ACT idioms when using `ConcurrentHashMap`. Figure 2 presents examples of code where the programmer meant to use `put-if-absent` semantics. Notice there are many variations. Figure 2(a) shows a temporary variable that is used to hold the result of the check. The check statement can use either `get` (Fig. 2(b)) or `containsKey` (Fig. 2(c)). Figure 2(e) and 2(f) show variations where the check condition is reversed. Notice that all of these variations have a `put` invocation that is control dependent on a `get` or `containsKey` invocation on the same map.

Next, we present one of the many atomicity violation scenarios that can occur in the examples in Fig. 2(a). We show it graphically in Fig 3. Suppose thread T_1 finds that the map does not contain the key, so it will calculate the value and try to put it into the map. Before T_1 puts, it is suspended and another thread T_2 puts a different value to the same key. Then T_1 resumes and executes its `put` operation. Under this scenario, the $\langle key, value \rangle$ pair put by thread T_2 will be overwritten by the `put` operation T_1. This violates the `put-if-absent` semantics of the original code.

Figure 4 and 5 show other misused CHECK-THEN-ACT idioms. Unlike Fig. 2 where we show many syntactic variations of the same idiom, in the subsequent figures, we only show one variation for each idiom.

Figure 4(a) shows that even when programmers use the new `putIfAbsent` operation instead of the old `put`, they still make mistakes. Notice that the code later uses the value that the programmer assumed to be mapped with the key. Now we describe an interleaving that results in an atomicity violation. After T_1 found that the map does not contain the

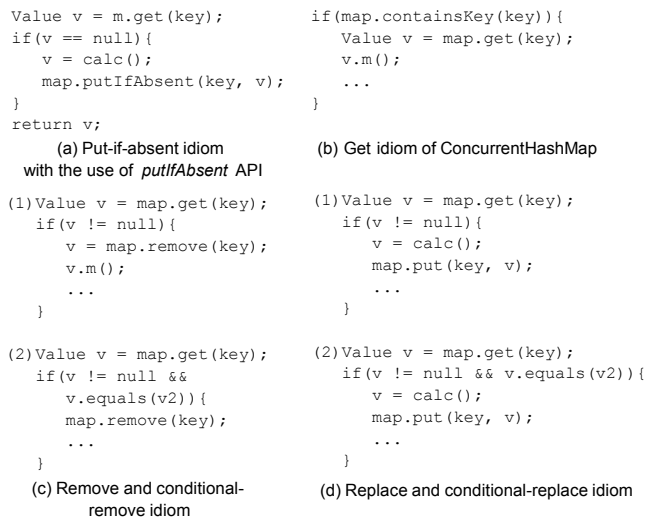


Figure 4. Other CHECK-THEN-ACT idioms for ConcurrentHashMap.

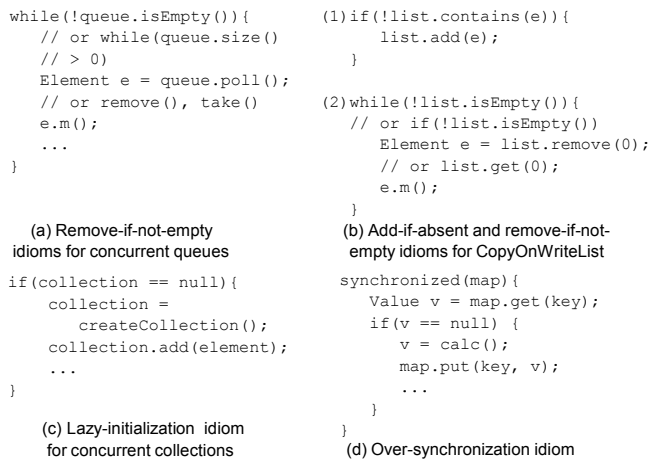


Figure 5. CHECK-THEN-ACT idioms of other types.

`key`, it calculates the value `v` and stores it to a reference that is later used. Before T_1 executes the `putIfAbsent` operation, thread T_2 puts another value to the same `key`. Then T_1 resumes, and its invocation of `putIfAbsent` will fail (since the `key` has been mapped by T_2). The last statement returns the reference to the stale value, which is not in the map.

Figure 4(b) shows an idiom involving the `get` operation. The code first checks that the map contains a given key, and then invokes a method on the value mapped to this key. An atomicity violation will occur when thread T_1 finds that the map contains the given key. Then T_2 removes the key, and subsequently, T_1 dereferences a `null` value. The code will throw a `NullPointerException`.

Figure 4(c) shows the idioms that remove elements. The first idiom (Fig. 4(c-1)) removes a $\langle key, value \rangle$ pair if the map contains the key, then subsequent statements use the removed value. Suppose thread T_1 finds that the map contains the key. Before it removes this $\langle key, value \rangle$, it suspends and T_2 removes the same pair. When T_1 resumes, its `remove` invocation returns a `null` value. Thus the subsequent statement that uses the value will throw a `NullPointerException`.

The second idiom (Fig. 4(c-2)) is a typical *conditional* removal. The code removes a $\langle key, value \rangle$ pair only if the key is mapped to a specific value `v2`. The atomicity violation occurs if T_2 puts another value (say `v3`) to the same key, after T_1 passed the check, but before it removed the pair. When T_1 resumes, the condition `v.equals(v2)` no longer holds, yet T_1 still removes the pair.

Figure 4(d) shows idioms that replace existing elements. These can be seen as complementary to `put-if-absent` semantics, since they have a `put-if-present` semantics. The atomicity violations will occur when thread T_2 removes the $\langle key, value \rangle$ pair while T_1 passed the check, and is about to perform the `put`. The second idiom (Fig. 4(d-2)) is a typical *conditional* replace operation.

B. *Misused* CHECK-THEN-ACT *in Queues*

The `j.u.c.` package contains several thread-safe implementations for working with queues. `ConcurrentLinkedQueue` is a traditional FIFO queue. Its queue operations do not block: if the queue is empty, the retrieval operation returns `null`. The package also provides `BlockingQueue`s to add blocking semantics to retrieval and insertion operations. If a queue is empty, the retrieval operation will block until an element is available.

Figure 5(a) shows the `remove-if-not-empty` semantics. The code first checks whether the queue contains some elements, and then it removes elements and uses them for further actions. Notice that there are several variations: the check statement can be an `if` or `while` statement, the check operation can query the size of the queue (e.g., `q.size() != 0` or `!q.isEmpty`) or `peek` inside to find elements. The act statement could use `poll, remove, take`, etc.

Here we describe one scenario for atomicity violation. Suppose the queue contains only one element and both threads T_1 and T_2 check the condition and find it is not empty. The thread that is the last to invoke the retrieval operation will get a `null` value which makes the code throw a `NullPointerException`.

C. *Misused* CHECK-THEN-ACT *in Lists*

The `j.u.c.` package contains a thread-safe implementation for working with lists. `CopyOnWriteArrayList` is a data structure in which all mutative operations (e.g., `add`) are implemented by making a fresh copy of the underlying array. Iterators iterate over a *snapshot* view of the collection at the point that the iterator was created.

Table I
THE SUMMARY OF CHECK-THEN-ACT IDIOMS. THE COLUMNS SHOW WHAT IS CHECKED, AND THE ROWS SHOW WHAT IS ACTED UPON.

Act \ Check	Reference	Object state
Reference	lazy-initialization (Fig. 5(c))	No examples
The state pointed by the checked object	non-null check [10]	put-if-absent (Fig. 2, 4(a)), get, remove, replace (Fig. 4(b), 4(c), 4(d)), add-if-absent(Fig. 5(b-1))
State other than the one checked	N/A	remove-if-not-empty (Fig. 5(a), 5(b-2))

Figure 5(b) shows two idioms. The first idiom (Fig. 5(b-1)) illustrates `add-if-absent` semantics. The code appends an element to a list, if the list does not already contain it. Two threads, T_1 and T_2 can both pass the check at the same time, and they will append the same element twice.

The second idiom (Fig. 5(b-2)) illustrates the `remove-if-not-empty` idiom, and the atomicity violation happens under the same interleaving as shown in Sec. II-B

D. Misused CHECK-THEN-ACT in Lazy Initialization

The `lazy-initialization` idiom is also error-prone. Figure 5(c) shows code that lazily creates a concurrent collection when it is needed. However, code also adds some elements into it. The atomicity violation will occur if both T_1 and T_2 find the collection reference is `null` and initialize it. In this case, one initialization will override the other. Now the elements added by T_1 are no longer seen by T_2.

E. Over-Synchronization in CHECK-THEN-ACT

Figure 5(d) shows a `put-if-absent` idiom wrapped by a synchronization block. Assuming that the other accesses to the map are protected by the same lock, this code is properly synchronized, thus the idiom executes atomically. However, the synchronization degrades the performance: it prevents threads who are working on different buckets of the map to operate in parallel. This defies the entire purpose of using a concurrent collection.

F. Summary of idioms

Based on the idioms we have described in previous subsections, we summarize the properties of the CHECK-THEN-ACT idioms that can lead to atomicity violations.

The `check` operation could query (i) the reference pointing to the collection (e.g., whether the reference is `null`), or (ii) the state of the collection (e.g., whether a `map` contains a given `key`).

The `act` operation could access (i) the reference pointing to the collection, (ii) the state of the collection w.r.t. the referenced object in the `check` (e.g., put a new $\langle key, value \rangle$ in the `map` using the previously checked `key`), or (iii) the state of the collection disregarding any particular object used in the `check` (e.g., removing all elements from a list).

```
Value v = map.get(key);
if(v == null) {
   v = calc();
   Value tmpV = map.putIfAbsent(key, v);
   if(tmpV != null)
      v = tmpV;
}
... // variable v is used here
```

(a) Fix for put-if-absent idiom

```
Value v = map.get(key);
if(v != null) {
   v = map.remove(key);
   if(v != null) {
      v.m();
      ...
   }
}
```

(b) Fix for remove idiom

```
Value v = map.get(key);
if(v != null) {
   v = calc();
   Value tmpV = map.
      replace(key, v);
   if(tmpV != null) {
      ...
   }
}
```

(c) Fix for replace idiom

```
Value v = map.get(key);
if(v != null) {
   v.m();
   ...
}
```

(d) Fix for get idiom

```
while(!queue.isEmpty()) {
   Element e = queue.poll();
   if(e != null) {
      e.m();
      ...
   }
}
```

(e) Fix for idiom of queue

Figure 6. Fixes for CHECK-THEN-ACT idioms.

Thus, there are 6 combinations of `check` and `act` operations. Table I groups all the previous idioms into these 6 combinations, using the above classification. Notice that one cell is not applicable, another cell is applicable - though we did not find examples in the projects that we studied, and for one cell we did not find examples, though there are examples in the literature [10] (e.g., if the object is not `null`, invoke a method on it).

Developers or researchers could use our Tab. I to manually look for CHECK-THEN-ACT atomicity violations in their code or to design bug detection tools. Though we have observed the check and act operations on instances of collections, similar operations can appear on arbitrary objects that are accessed concurrently.

G. Correction

We can use two ways to correct the atomicity violations caused by misused CHECK-THEN-ACT idioms: (1) leveraging the proper atomic API provided by the concurrent collections, or (2) adding a synchronization block around the CHECK-THEN-ACT code.

Figure 6 shows the strategies that we use to fix the misused CHECK-THEN-ACT idioms. We underlined the statements that we add or change. For the idioms that have `put-if-absent` semantics, we use the `putIfAbsent` operation instead of `put`. When the code further reads the value placed in the map, our fix ((Fig. 6(a)) checks the status of the `putIfAbsent` to judge whether the assumed value

was indeed placed in the map (`putIfAbsent` returns `null` to indicate successful execution). Note that for `put-if-absent` idiom with the use of `putIfAbsent` method, our fix also checks the status of the `putIfAbsent`.

In the fixes in Fig. 6(b), 6(c) and 6(e), CTADETECTOR adds code to check the return value of the `act` operation, thus preventing `NullPointerException`s. For the `get` idiom in Fig. 4(b), we replaces the use of `containsKey` with checking whether the mapped value is not `null`.

Note that we do not show the fixes for the `add-if-absent` and `lazy-initialization` idioms. The fix for the former is similar to `put-if-absent`, while the fix for the latter is wrapping the idiom with a proper synchronization block. To fix the performance bugs because of over-synchronization, CTADETECTOR removes the lock and uses the corresponding compound update API method. For example, in Fig. 5(d), CTADETECTOR removes the synchronization and uses `putIfAbsent` instead of `put`.

III. ANALYSIS OF IDIOM INSTANCES

In this section we answer four research questions:

- **RQ1:** What are the commonly used CHECK-THEN-ACT idioms in real-world programs?
- **RQ2:** Which idioms are the most error-prone?
- **RQ3:** Do misused idioms result in real bugs? Are our patches accepted by developers?
- **RQ4:** What is the evolution of programs w.r.t. CHECK-THEN-ACT idioms?

RQ1 and RQ2 help us, library designers, and tool builders learn about the state of the practice. RQ3 evaluates whether the found misused idioms are critical for the correctness or performance of real world programs. RQ4 shows whether developers pay more attention to CHECK-THEN-ACT idioms.

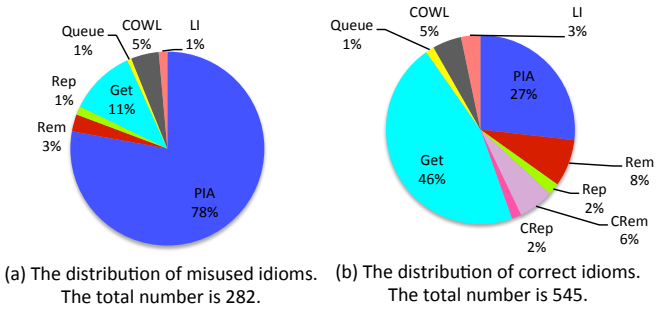
A. Experimental setup

Subjects: To answer the first three research questions, we used a corpus of 28 real-world open-source programs. The first three columns of Table II show the subject programs, the version number, the size – in non-blank, non-comment source lines of code (SLOC)[1], and the domain of application. All programs use concurrent collections. For each program, we use the most current version.

To study the evolution of the programs (RQ4), out of the initial corpus, we selected those projects that had multiple releases between 2007 and 2012. This created a corpus of 18 programs. For each program, we chose three major releases: V_3 – the most current release (as shown in Tab. II), V_2 – a major release from 2010–2011, and V_1 – a major release from 2007–2009.

Process: We ran our tool, CTADETECTOR, over our corpus. CTADETECTOR classified idioms as correct or misused. The latter contains semantic or performance issues. We manually verified the results and sorted them based on the idioms that we introduced in Sec. II.

[1]as reported by the SOURCECOUNTER [11] tool



(a) The distribution of misused idioms. The total number is 282.

(b) The distribution of correct idioms. The total number is 545.

Figure 7. The distribution of idioms. (PIA: `put-if-absent`, Rem: `remove`, Rep: `replace`, CRem: `conditional-remove`, CRep: `conditional-replace`, Get: `get`, Queue: idioms for queues, COWL: .idioms for lists, LI: `lazy-initialization`)

To confirm whether the misused idioms result in real bugs, we reported 155 instances to the open-source developers. Our companion website [12] contains links to our bug reports. Along with the bug description, we also submitted a patch generated by CTADETECTOR. When developers reported that a misused idiom does not result in a real bug, we further asked them to elaborate why the atomicity violation in the idiom is acceptable for their program.

To answer the evolution question we compare the number of correct and misused instances of idioms along the three major releases.

B. Results

RQ1: What are the commonly used CHECK-THEN-ACT idioms in real-world programs?

Fig. 7 shows the distribution of correct and misused idioms across the corpus of 28 projects. CTADETECTOR found 282 instances of misused idioms and 545 instances of correct idioms.

Notice that in each category, there is one idiom that clearly dominates the others: the `put-if-absent` idiom is the most common misused idiom, while `get` is the most common correctly used idiom. It is also surprising that the top four idioms in each category are different.

Our result shows that 93% (259) of the misused instances and 90% (492) of the correct instances appear when using `ConcurrrentHashMap`. This is expected: (i) a previous study [8] shows that `ConcurrrentHashMap` is the most widely used concurrent collection in Java, and (ii) `ConcurrentHashMap` stores $\langle key, value \rangle$ pairs so it offers a richer API than other collections, thus there are more choices to compose operations.

RQ2: Which idioms are the most error-prone?

Columns 5–13 in Tab. II show the number of misused and correct instances of idioms for each program. With

Table II
CHECK-THEN-ACT IDIOMS IN REAL-WORLD PROGRAMS.

Subject Name	SLOC	Description	PIA		Rem		Rep		CRem		CRep		Get		Queue		COWL		LI	
			m	c	m	c	m	c	m	c	m	c	m	c	m	c	m	c	m	c
Annsor 1.0.3	1605	Annotation runtime processor	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Apache Cassandra 1.1.1	132183	Distributed database	3	5	-	1	-	-	-	-	-	-	2	8	1	1	-	-	-	-
Apache CXF 2.6.1	441269	Open source services framework	5+4	9	-	1	-	-	-	1	-	-	3	8	-	1	2	6	1	2
Apache Lucene 4.0.0	361494	Text search engine library	1+2	1	-	-	-	-	-	-	-	-	-	4	-	-	-	-	-	-
Apache Mina 2.0.4	46435	Network application framework	2+2	2	-	-	1	-	1	-	1+1	-	-	5	-	1	-	-	-	-
Apache Struts 2.3.4	146919	Web applications framework	5+1	2	-	-	-	1	-	3	-	1	-	1	-	-	1	-	-	-
Apache Tomcat 7.0.28	215298	Servlet container	2	5	-	-	-	-	-	-	-	-	2	10	-	2	-	-	1	-
Apache Trinidad 2.0.1	220484	JSF framework	14+2	5	-	2	0+1	-	-	1	-	-	1	8	-	-	-	-	-	4
Apache Wicket 1.5.7	169142	Java web framework	10	3	-	-	-	-	-	-	-	-	-	7	-	-	1	-	-	-
BlazeDS 4.0.1	68887	Web messaging technology	1+3	2	2	3	-	-	-	-	-	-	6	11	-	-	1	16	-	9
Carbonado 1.2.3	54254	Persistence abstraction layer	2	2	-	-	-	-	-	2	-	-	-	2	-	-	-	-	-	-
CBB 1.0	17001	Concurrent Building Blocks	4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
DWR 1.1	35630	Ajax for Java	1	10	-	1	-	-	-	-	-	1	-	3	-	-	-	-	-	-
Ektorp 1.1.1	10112	Java API for CouchDB	4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Flexive 3.1.6	139011	Content management system	10+2	8	-	-	-	4	-	-	-	1	2	2	-	-	0+1	-	-	-
Glassfish 3.1	721944	Application server	5+6	11	2	7	-	-	-	1	-	-	2	33	-	1	1	-	1	1
Granite 2.3.2	41790	Data Service	5+2	14	-	1	-	-	-	-	-	-	-	10	-	-	-	-	-	-
Hazelcast 2.0.4	89080	Data distribution platform	14+3	13	-	4	-	-	-	3	-	-	2	39	-	-	-	-	-	-
Ifw2 1.33	55596	Web application framework	2	5	-	-	-	-	-	-	-	-	2	1	-	-	-	-	-	-
JBoss AOP 2.2.2	196106	Aspected oriented framework	11+6	11	-	-	-	-	-	-	-	-	-	10	-	-	-	-	-	-
JSefa 0.9.3	18173	Object serialization library	1+2	3	-	-	-	-	-	-	-	-	-	2	-	-	-	-	-	-
Memcache	6695	Memory object caching system	6	-	-	1	-	-	-	-	-	1	2	8	-	-	-	-	-	-
Open EJB 4.0.0	286451	EJB container	6+5	2	2	4	-	-	-	2	-	-	4+1	5	-	-	-	-	-	-
Open JDK 8	2262000	Java development kit 8	19+3	24	-	8	-	6	-	16	-	5	-	19	1	-	1	2	1	-
RestEasy 2.3.4	123813	JAX-RS client framework	11	4	-	-	-	-	-	2	-	-	-	9	-	-	-	-	-	-
Tersus	113260	Visual programming platform	1	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Vo Urp	29954	Data models translator	3	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Zimbra	448573	Collaboration server	14+12	3	2	11	0+2	-	-	2	-	-	0+2	44	-	2	3+2	3	-	2
Total	**6453159**		**163+55**	**146**	**8**	**44**	**1+3**	**11**	**1**	**33**	**0+1**	**9**	**28+3**	**249**	**2**	**8**	**10+3**	**27**	**4**	**18**

For each idiom, the column `m` and `c` represent the misused and correctly used instances. For the misused instances, the number on the left of plus sign shows semantic issues, whereas the number on the right shows performance bugs (where applicable). The columns 4 to 12 represent `put-if-absent`, `remove`, `replace`, `conditional-remove`, `conditional-replace`, `get` idioms, idioms for queues, `CopyOnWriteList` and `lazy-initialization`.

the exception of the `put-if-absent` idiom, notice that the number of correct instances outweighs the misused instances to a large extent for most projects. This means most developers are aware of how to correctly use the concurrent collections, but may make mistakes occasionally.

For `put-if-absent` idiom, the number of misused instances is larger than the number of correct instances. This shows that this idiom is the most error-prone. However, as we show in RQ3, not all misused instances are perceived as buggy by the developers.

For the misused idioms, we show the number of instances as the sum of two numbers: the first shows semantic bugs, the second shows performance issues. Tab. II shows 65 instances of idioms where the developers wrapped the CHECK-THEN-ACT within a synchronized block. However, this is over-synchronization, and we classify these instances as performance bugs. This shows that some developers know there are atomicity violations in the idioms, but they add synchronization to avoid them, instead of using the atomic APIs from the concurrent collections. In contrast, CTADETECTOR correctly suggests patches that involve the atomic APIs, as discussed in Section II-G. This can dramatically improve the performance.

RQ3: Do misused idioms result in real bugs? Are our patches accepted by developers?

In our corpus of projects, we selected the 17 most active projects. We reported the misused idioms and also provided the patches generated by CTADETECTOR. For some large projects like GlassFish, we did not report all the misused idioms that are detected by CTADETECTOR, but only those for the major components.

For the 17 projects that we contacted so far, we reported 155 bugs. However, we only got replies from the developers of 11 projects. Table III shows these 11 projects, along with the number of bugs we reported and were replied in each project (column 2), and the number of bugs confirmed by developers (column 3). The bugs that are confirmed include

Table III
BUG CONFIRMATION FROM THE DEVELOPERS: 49 SEMANTIC BUGS AND 11 PERFORMANCE BUGS.

Subject Name	Replied Bugs	Confirmed	Fixed Version
Apache Cassandra	6	5	1.1.2
Apache CXF	15	15	2.7.0
Apache Mina	5	5	2.0.5
Apache Struts	7	2	2.3.5
Apache Tomcat	5	4	7.0.30
Apache Trinidad	1	0	-
Apache Wicket	11	11	1.5.8
Glassfish	8	6	4.0
GraniteDS	7	7	3.0.0 beta1
OpenJDK	14	0	-
RestEasy	11	5	2.3.5
Total	**90**	**60**	

49 semantic and 11 performance bugs. The developers of 9 projects accepted our patches and included the patches in the new versions. Last column shows the version numbers that include our patches.

As shown in Table III, not all of the misused idioms lead to bugs, although two thirds of the instances cause buggy behaviors in the programs. For the remaining one third of our reported misused idioms, the developers do not think these result in buggy behaviors.

The reasons that the developers provided can be divided into three categories:

1. Impossible interleaving: The buggy interleaving that we described in Section II does not happen in the application context. This can be due to two reasons. First, the code containing the idiom is never executed concurrently (e.g., this was one case in Cassandra). This was surprising to us, since this defies the whole reason of using a concurrent collection. However, it could be that only some code snapshots that use an instance of a concurrent collection are executed concurrently, or it could be that developers envision some future evolution where the code will indeed run concurrently. Second, the conflicting operation never executes concurrently. For example, in `Tomcat`, in one `put-if-absent` instance, at any given moment, there is only one thread that puts a value in the map.

2. Unique values: For some `ConcurrentHashMap` usages, the program *uniquely* calculates one single `value` for a given `key`. That is, the `value` is either a singleton object [9], or the program can calculate several value objects for the same key, but they are in the same equivalence class. Thus, for the `put-if-absent` idiom, even if the value written by one thread is overwritten by another thread, since the two values are equivalent, the idiom does not lead to bugs. In `Open JDK 8`, there are 13 cases when the values are uniquely calculated from the keys.

3. Program resilience: The program does not care whether a value written by one thread is overwritten by another thread. For instance, in `Apache Struts`, there is one case when the `ConcurrentHashMap` is used as a cache. Even if the value is overwritten and no longer in the map, it can still be used without affecting the behavior. For `lazy-initialization` idiom, there is a case in `GlasshFish` where even if the values put into the map are lost, those values will be created and put again by other threads.

Table IV
THE EVOLUTION OF IDIOMS

Instances \ Version	V_1	V_2	V_3
misused (m)	112	156	201
correct (c)	152	253	418
m/(m+c)	**0.42**	**0.38**	**0.32**

Discussion: In the above cases, the race conditions in the idioms are benign and can improve the performance (e.g., `put` is faster than `putIfAbsent`). Notice that reasoning about such cases requires *deep understanding* of the domain and the concurrency model of the program. This is usually beyond the capabilities of tools and is better left to human expert judgement. This is exactly the reason why CTADETECTOR is *interactive*, allowing the human expert to judge whether the misused idiom is really a bug.

However, developers should carefully check the semantics of the programs to make sure they use an idiom correctly, since as our result shows, 67% of misused instances lead to real bugs. Furthermore, the developers should document the invariants that ensure correctness. This can prevent future versions running afoul precisely because of these bugs. In the 30 instances that developers did not considered real bugs, they documented only one such invariant.

RQ4: What is the evolution of programs w.r.t. CHECK-THEN-ACT idioms?

For the 18 projects that have multiple major releases, Table IV shows the total number of instances of idioms across three major releases. Notice that the number of instances increases for both misused and correctly used idioms. This means that developers are embracing concurrent collections. This is consistent with our recent finding [5] that shows that many developers are embracing multicore parallel programming.

Interestingly, the ratio of misused instances (as shown by the last row) decreases in later versions. This shows that developers pay more attention to the correct usage of CHECK-THEN-ACT idiom. Possible explanations are that as time goes by, programmers have more resources to learn how to use the concurrent collection correctly, or they found such bugs in production.

IV. ANALYSIS INFRASTRUCTURE

In this section we describe our approach to automatically detect and correct the CHECK-THEN-ACT idioms that we listed in Sections II–III. Subsection IV-A presents the detection and correction approach.

A. Idiom detection and correction

We implemented both the detection and correction in a tool, CTADETECTOR, on top of Eclipse Java development tools (JDT) [13]. When CTADETECTOR finds a match between the source code and the idioms, it reports the detected idiom as well as the source code location.

To detect idioms, we employ a static code analysis that uses syntactical and semantical information to match conditional statements from the source code of a program to the idioms we presented in Section II.

The analysis visits all the conditional statements (i.e., `if` and `while`) in a program. For each conditional statement, the analysis iterates over all the idioms and tries to determine a match. To determine a match, the analysis needs to verify whether: (i) the conditional expression matches the `check` part of the idiom, (ii) the conditional statement operates over an instance of a concurrent collection, and (iii) the body of the conditional statement matches the `act` part of the idiom.

Next, we illustrate how the analysis matches one of the idioms, namely the `put-if-absent` from Fig. 2(e). First, the analysis checks the expression used in the `if`'s condition. This means determining whether (a) the code invokes the `containsKey` (b) the condition is negated.

Second, the analysis checks whether `if` statement operates over an instance of `ConcurrentHashMap`. To do this, the analysis gets the type information of a variable from the static type binding (this determines that the variable is an instance of `Map`) and the variable initialization statement (this determines that the map variable is initialized with a `ConcurrentHashMap`). Note that we use an inter-procedural analysis to find out whether a variable is initialized with a concurrent collection.

Third, the analysis checks whether (a) the body statements invoke the `put` method (b) the `put` is invoked on the same `ConcurrentHashMap` object used in the condition expression, and (c) it places in the map the same `key` object that was used in the condition expression.

To correct the reported misused idioms, CTADETECTOR uses the fixes that we presented in Subsect. II-G. We implemented the correction on top of Eclipse's AST rewriting engine. Notice that we take an *interactive* approach: the programmer can inspect the report, and if she agrees that it is indeed a problem, she can choose to apply the correction transformation that CTADETECTOR suggests. For each suggested transformation that tool shows a preview of the code before and after the transformation. The companion website shows screenshots.

B. Discussion

Despite the fact that our approach is pattern-based, it is quite effective and efficient. Here we discuss several potential improvements, that we decided not to include since they will not necessarily improve the analysis.

1. CTADETECTOR only performs an intra-procedural idiom matching, thus it may miss cases when the check and act operations are in different methods. For example, for the idiom shown in Fig. 2(e), the `if(!map.containsKey(key))` and `map.put(key, v)` may be in different methods. However, in the 28 projects we used in our empirical evaluation, we manually found only one single case (in Apache Mina) that needs inter-procedural analysis. That means intra-procedural analysis can detect most of the misused idioms.

2. CTADETECTOR uses static type binding information collected at compile time to determine whether a variable represents a concurrent collection object or whether two arguments are the same. However, using the static type binding information can be inaccurate since the variables may be reassigned and point to different objects between check and act operations. In this case, we need points-to analysis to determine whether two variables point to the same object. However, in the 28 projects we used, there is only one case (in OpenJDK 8) in which a variable may either point to a `HashTable` or a `ConcurrentHashMap`, depending on some conditions. In all other cases, a local variable or a field used in check or act operations is never changed. Thus, using points-to analysis will only have modest improvements.

V. RELATED WORK

We organize the related work into: (i) empirical studies for concurrent programming, (ii) detection of atomicity violations, and (iii) pattern-based program analysis.

Empirical study for concurrent programs. Lu et al. [10] categorized concurrency bug types by analyzing a large number of bug reports from open-source repositories. They list one of the six types of atomicity violations that we classify in Tab. I. In a followup work [14] they also describe bugs that manifest as performance slowdowns in concurrent programs. Schaefer et al. [15] showed several examples of how sequential refactorings can break concurrent programs.

We have previously conducted an empirical study [5] on how developers from thousands of open-source projects use Microsoft's Parallel Libraries. One of the findings was that some library constructs are error-prone. Also, our previous work [16] on automated refactoring to introduce concurrent library constructs showed that manual refactorings from `HashMap` to `ConcurrentHashMap` are error-prone.

Our current work focuses on the study of how programmers misuse concurrent collections.

Atomicity checking techniques. Several researchers proposed dynamic [6], [17]–[20] or static techniques [21]–[23] to check atomicity violations in concurrent programs. Some approaches require programmers to provide test drivers, but constructing test drivers for large applications is time consuming. Others require programmers to write annotations, but industry programmers are reluctant to write annotations.

Among these techniques, COLT [6] is a recent dynamic tool that checks the atomicity of composed operations from

Java concurrent collections. COLT found 41 problematic atomicity violations in 25 open-source projects. For the same projects used in COLT's evaluation, CTADETECTOR found 178 violation instances, from which we reported 85, and 55 of them are confirmed to be new bugs.

Pattern-based analysis. Pattern inference and identification is also a widely used approach to improve software quality. `AVIO` [24] and `Falcon` [25] analyze the access patterns of variables to detect or locate concurrency bugs. `FindBugs` [26] detect bugs by statically matching the bug patterns to programs. The current version of `FindBugs` only considers one single variation out of our nine idioms. Yu et al. [27] exploit interleaving idioms to test concurrent programs. Uddin et al. [28] infer temporal API usage patterns that can be used to improve the API design and usage. Wendehals and Orso [29] proposed a dynamic technique to recognize design patterns in the programs. However, our work focuses on the patterns of concurrent collection usage.

VI. CONCLUSIONS

Some programmers erroneously think that just by using thread-safe concurrent collections their code is thread-safe. Our study of 28 projects reveals nine common CHECK-THEN-ACT idioms that can result in atomicity violations. We found that the distribution of correct and misused idioms is not the same, which means that some idioms are more error-prone than others. This finding is important for library designers who can design more resilient APIs. It also provides educational value for developers who use concurrent collections.

Using this corpus and our tool, CTADETECTOR, we found 282 buggy instances. The developers examined 90 of them and confirmed 60 as new bugs, and applied our patch. While they confirmed 67% of the examined bugs, they claim that the remaining do not result in bugs. This reasoning requires deep understanding of the domain and concurrency model. We hope that our study motivates other follow-up studies to fully understand these bugs and eradicate them.

Acknowledgments: We would like to thank Darko Marinov, Milos Gligoric, Stas Negara, Semih Okur, Cosmin Radoi, Caius Brindescu, Mihai Codoban, Shanxiang Qi, Wonsun Ahn, and the anonymous reviewers for their feedback on earlier versions of this paper. This research is partly funded through NSF CCF-1213091 and CCF-1219027 grants, a gift grant from Intel, and the Intel-Illinois Center for Parallelism at the University of Illinois at Urbana-Champaign. The Center is sponsored by the Intel Corporation.

REFERENCES

[1] "Task Parallel Llibrary (TPL)," http://msdn.microsoft.com/en-us/library/dd460717.aspx.
[2] "Collections.Concurrent (CC)," http://msdn.microsoft.com/en-us/library/dd997305.aspx/.
[3] "Threading Building Block (TBB)," http://threadingbuildingblocks.org.
[4] "Java Concurrent Library," http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html.
[5] S. Okur and D. Dig, "How do developers use parallel libraries?" in *FSE*, 2012.
[6] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav, "Testing atomicity of composed concurrent operations," in *OOPSLA*, 2011.
[7] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes, *Java Concurrency in Practice*, 2005.
[8] W. Torres, G. Pinto, B. Fernandes, J. a. P. Oliveira, F. A. Ximenes, and F. Castor, "Are Java programmers transitioning to multicore? A large scale study of Java FLOSS," in *TMC*, 2011.
[9] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides, "Design patterns: Abstraction and reuse of object-oriented design," in *ECOOP*, 1993.
[10] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *ASPLOS*, 2008.
[11] "SourceCounter," http://code.google.com/p/boomworks/wiki/SourceCounterEN.
[12] http://mir.cs.illinois.edu/~yulin2/CTADetector.
[13] "JDT," http://www.eclipse.org/jdt/.
[14] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *PLDI*, 2012.
[15] M. Schäfer, J. Dolby, M. Sridharan, E. Torlak, and F. Tip, "Correct refactoring of concurrent java code," in *ECOOP*, 2010.
[16] D. Dig, J. Marrero, and M. D. Ernst, "Refactoring sequential Java code for concurrency via concurrent libraries," in *ICSE*, 2009.
[17] D. Weeratunge, X. Zhang, and S. Jaganathan, "Accentuating the positive: atomicity inference and enforcement using correct executions," in *OOPSLA*, 2011.
[18] C.-S. Park and K. Sen, "Randomized active atomicity violation detection in concurrent programs," in *FSE*, 2008.
[19] W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," in *ASE*, 2000.
[20] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *OSDI*, 2008.
[21] C. Flanagan and S. Qadeer, "A type and effect system for atomicity," in *PLDI*, 2003.
[22] C. von Praun and T. R. Gross, "Static detection of atomicity violations in object-oriented programs," in *Journal of Object Technology*, 2003.
[23] M. Vaziri, F. Tip, and J. Dolby, "Associating synchronization constraints with data in an object-oriented language," in *POPL*, 2006.
[24] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "Avio: detecting atomicity violations via access interleaving invariants," in *ASPLOS*, 2006.
[25] S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: fault localization in concurrent programs," in *ICSE*, 2010.
[26] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in *OOPSLA*, 2004.
[27] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A coverage-driven testing tool for multithreaded programs," in *OOPSLA*, 2012.
[28] G. Uddin, B. Dagenais, and M. P. Robillard, "Analyzing temporal API usage patterns," in *ICSE*, 2011.
[29] L. Wendehals and A. Orso, "Recognizing behavioral patterns at runtime using finite automata," in *WODA*, 2006.