Verifying Atomicity via Data Independence

Ohad Shacham
Yahoo Labs, Israel

Eran Yahav
Technion, Israel

Guy Golan-Gueta
Yahoo Labs, Israel

Alex Aiken
Stanford University, USA

Nathan Bronson
Stanford University, USA

Mooly Sagiv
Tel Aviv University, Israel

Martin Vechev
ETH Zurich, Switzerland

ABSTRACT

We present a technique for automatically verifying atomicity of composed concurrent operations. The main observation behind our approach is that many composed concurrent operations which occur in practice are *data-independent*. That is, the control-flow of the composed operation does not depend on specific input values. While verifying data-independence is undecidable in the general case, we provide succint sufficient conditions that can be used to establish a composed operation as data-independent. We show that for the common case of concurrent maps, data-independence reduces the hard problem of verifying linearizability to a verification problem that can be solved efficiently with a bounded number of keys and values.

We implemented our approach in a tool called VINE and evaluated it on *all* composed operations from 57 real-world applications (112 composed operations). We show that many composed operations (49 out of 112) are data-independent, and automatically verify 30 of them as linearizable and the rest 19 as having violations of linearizability that could be repaired and then subsequently automatically verified. Moreover, we show that the remaining 63 operations are not linearizable, thus indicating that data independence does not limit the expressiveness of writing realistic linearizable composed operations.

Categories and Subject Descriptors:

D.2.4 [**Software/Program Verification**]; D.1.3 [**Concurrent Programming**]

General Terms:

Verification

Keywords:

concurrency, linearizability, verification, composed operations, data-independence, collections

ISSTA'14, July 21–25, 2014, San Jose, CA, USA
Copyright 2014 ACM 978-1-4503-2645-2/14/07...$15.00
http://dx.doi.org/10.1145/2610384.2610402

```
Attribute removeAttribute(String name) {
  Attribute val = attr.get(name);
  if (val != null) {
    val =  attr.remove(name);
  }
  return val;
}
```

Figure 1: **Linearizable version of a composed operation taken from Apache Tomcat [4]. This operation is not supported by the enriched Java library [1]. This composed operation's semantics is equivalent to `attr.remove(name)`, however, it is more efficient in the general case due to Java's ConcurrentHashMap internal locking mechanism.**

1. INTRODUCTION

Modern programming languages such as Java and C# provide efficient concurrent library implementations that allow developers to build scalable applications. The basic intention is to free the programmer from having to deal with complex concurrency issues by providing a well-defined interface to commonly used operations.

However, applications using modern libraries often require functionality which involves a *combination* of several library operations. This functionality is typically achieved by building a new operation which is intended to behave atomically, with the new operation built of several existing library operations. Ironically, well-designed libraries are more likely to increase the number of complex composed concurrent operations as more applications are using the library.

Unfortunately, building correct and efficient composed concurrent operations is tricky and error-prone. A recent study of composed operations found in real-world applications reported a large number of atomicity violations [27]. This study influenced the interface of Java concurrent collections, which are being enriched to avoid common programming pitfalls [1, 2]. While the library interface can be extended to support certain common composed operations, it is impossible to include all possible combinations (the set is potentially infinite). Hence, despite the best of intentions by library designers, the fundamental problem of building correct and efficient application-specific composed operations is unlikely to go away. Indeed, the example shown in Figure 1, taken from Apache Tomcat [4] is a custom composed operation that is atomic and is unsupported by the enriched library [1].

Our Approach.

We have built a tool, called VINE, which verifies atomicity (linearizability) of composed operations. For instance, our tool verifies that the operation in Figure 1 is linearizable. Our tool enables programmers to use the existing library interface by ensuring that the composed operation is linearizable. We show that for most practical composed operations, our tool is able to verify the correctness of the composed operation in seconds. This is surprising as checking linearizability is generally considered a hard problem [11].

The key observation behind our approach is that in practice, many composed concurrent operations are *data-independent*. That is, the control-flow of the composed operation is independent of specific input values. While verifying data-independence is generally undecidable, we provide succinct sufficient conditions which ensure that the composed operation is data-independent.

Further, we conducted an empirical study using code search engines [8] and discovered that the majority of composed operations are based on concurrent maps. To address this case, we present a novel reduction-based verification approach for linearizability of data-independent composed operations build on-top-of concurrent maps. The basic idea of our approach is to show a reduction from an operation that can use arbitrary keys and values to an operation that uses a *bounded* number of keys and values. That is, if we verify the operation with some keys and values, our method guarantees that it will be correct for any keys and values. The actual bounded verification of linearizability for data-independent composed operations can be done using standard techniques (e.g. [27]).

Evaluation.

We used static analysis to extract composed operations from applications and prove that they are data-independent. Our linearizability verifier is implemented in Promela, and verifies the extracted operations using the SPIN model checker [24]. We also implemented the composed operations extractor using the WALA [10] static analyzer. We extracted 112 composed operations from 57 applications and atomatically checked which of these satisfy the data-independence requirement. We found that 63(56%) of these are not data-independent and cannot be handled with our technique. However, a close inspection revealed that these 63 operations are in fact non-linearizable! Of the 49 operations that can be handled by our technique, 44 are data-independent and 5 depend on fixed input keys. SPIN checked the extracted models for these 49 operations in less than a second each, verifying correctness of 30 operations and identifying a linearizability violation in the remaining 19 operations. We manually fixed the linearizability violations in these 19 operations and our tool then successfully proved all of the repaired operations linearizable.

Contributions.

The contributions of this paper are as follows:

- We define a notion of data-independence for composed operations. This notion is inspired by Wolper [31] and can be established via a simple static analysis.
- We conduct a thorough empirical study proving that many realistic composed operations are indeed data-independent and all of the data-dependent composed operations are non-linearizable.
- We proved that for data-independent composed operations, checking linearizability can be performed by using a single key and two values. In particular, for programs with finite local state, verifying linearizability becomes decidable.
- We show that in practice checking linearizability for data-independent composed operations can be performed efficiently (in about a second).

The rest of this paper is organized as follows: Section 2 motivates the problem by showing a linearizable data-independent composed operation taken from [3]. Section 3 describes a data centric checking for linearizaibility of composed operations. Section 4 defines data-independent composed operations and shows that proving linearizability of data-independent composed operations can be done with fixed map entries. Section 5 describes our implementation and a classification of real-world composed operations. Section 6 reviews related work and Section 7 concludes.

2. MOTIVATION

Figure 2 shows a composed operation taken from Apache ServiceMix [3] named `getLock`. This operation uses the underlying `ConcurrentHashMap locks` to memoize the `ReentrantLock` allocated at line 4. When the value for a given `id` is cached in the collection it is returned immediately; when the `ReentrantLock` for a given `id` is not available, it is allocated and inserted into the collection. This function handles concurrency in an optimistic manner. The code in lines 5–8 guarantees that even if the update fails when other threads succeed in the update, the return value of the operation is consistent.

This implementation is correct but tricky. It is worth noting that this implementation is not conflict-serializable [20] since there exists an execution of the operation with two memory accesses at lines 2 and 6 and an intervening operation by another thread that mutates the environment.

The method `getLock` appears to occur automically to clients i.e., it is linearizable according to [23]. There are two cases that can occur. The first case, handled on line 2, is when `locks.get(id)` returns a non-`null` value ("id" is in `locks`) as then `getLock` returns the value extracted from `locks`. In the second case, handled at line 5, if `putIfAbsent` returns `null` (indicating the update succeeded) then `lock` is returned, and otherwise, the current collection value corresponding to `id` is assigned to `oldLock`, which is then assigned to `lock` at line 8 and returned. Checking any safety properties of such operations is potentially hard due to the unbounded number of potential collection values. Moreover, concurrency drastically increases the state space that must be considered.

Interestingly, we observed that composed operations in the real-world follow a common control structure. Indeed, these are "essentially finite-state programs" in the sense that linearizability can be proven by investigating a fixed number of input values. Next, we briefly summarize the main ideas which enable verification of linearizability for composed operations.

Data Independent Composed Operations.

We first formulate the notion of *data-independence*, inspired by Wolper [31]. Informally, Wolper requires that the control-flow of the composed operation does not depend on its input values and provides syntactic notions which guarantee data-independence. Unfortunately, in practice, Wolper's notion of data-independence is too restrictive for composed operations. An analysis of 112 composed operations shows that none of them are data-independent according to Wolper's definition.

To address this issue, we define a notion of data-independence which takes into account the meaning of the collection (i.e., a map). We define a class of composed operations such that their control flow is independent of the actual collection values being manipulated. We refer to this class of programs as **S**ingleton **C**ollection **M**ethods (SCM). The operation `getLock` is an example of an SCM: it takes as input an `id`, which is used only as a key inside the `locks` collection. The return value from the `locks` operations is checked in `getLock` only for equality with `null`, meaning whether there

```
1 public Lock getLock(String id) {
2  Lock lock = locks.get(id); // @LP lock!= null
3  if (lock == null) {
4   lock = new ReentrantLock();
5   Lock oldLock =
6     locks.putIfAbsent(id, lock); // @LP
7   if (oldLock != null) {
8    lock = oldLock;
9   }
10 }
11 return lock;
12 }
```

Figure 2: Linearizable data-independent composed operation taken from Apache ServiceMix [3]

exists a key `id` inside `locks`. Further, all branches are of the same kind (equality with `null`). Note that according to Wolper, this operation is not data-independent since its control flow is dependent on the results of operations at lines 3 and 7.

Bounding the number of inputs.
We prove that data-independent composed operations that are linearizable w.r.t one input key must be linearizable with respect to any input key. In `getLock` this means that if `getLock` is linearizable for a specific `id`, then it is linearizable for *any* input `id`. Thus, the specific choice of input key is irrelevant to the verification. Our results also extend to operations which depend on a fixed number of input keys.

Bounding the global state size.
Finally, we also prove that the number of map entries needed to expose linearizability violations in data-independent composed operations is bounded. Thus, for programs with bounded local states such as `getLock`, checking linearizability is actually decidable even though it potentially manipulates unbounded inputs (`String id`), number of map entries, and number of threads.

3. THREAD-CENTRIC LINEARIZABILITY CHECKING

In this section, we describe a simple thread-centric procedure for checking linearizability of composed concurrent operations, taken from [27].

A *local state* $l \in \mathcal{L}$ describes the values of the local variables. For simplicity we assume that there is an extra local variable pc denoting the current program location. We also assume that the stack is part of the local state.

Let $\mathcal{C}$ be the set of base collection values and let $\mathcal{A}$ be the set of values for actual arguments and return values of the collection. Let $\Sigma = \mathcal{L} \times \mathcal{C}$ denote the set of program states.

Let C.M be a composed operation in the custom map C. We say that a method of C is a base method if it is not C.M. We use $Stmt$ to denote the set of possible program statements.

3.1 Transition Relation

Given a method of interest C.M, we define the transition relation $\Pi \subseteq \Sigma \times Stmt \times \Sigma$ as a small-step operational semantics. We define two kinds of transitions:

- Local Transition:

$$\langle l, c\rangle \xrightarrow{s} \langle l', c\rangle$$

  That is, executing a local statement in C.M on state $\langle l, c\rangle$ yields a state $\langle l', c\rangle$. It is essential that the stack be part of the local state. Here, s can be any statement except those that invoke C methods.

- Collection Transition: This transition always invokes base methods of C. We assume that invocation (calling and returning) of a base method is atomic. Depending on how the local state is affected and who performs the transition, we define two kinds of collection transitions:

  – Main Transition:

$$\langle l, c_i\rangle \xrightarrow{\texttt{x=C.b(args)}} \langle l[pc \mapsto q, x \mapsto r], c_o\rangle$$

  Here, `x=C.b(args)` is always a statement of C.M at $\langle l, c_i\rangle$ and the statement is followed by a label q in C.M. Thus, executing the transition from state $\langle l, c_i\rangle$ involves: (i) evaluating `args` to a in l; (ii) invoking method `b` with a, which produces a new collection c_o; and (iii) setting return value $r \in \mathcal{A}$ to the local variable `x`. Note that main transitions always invoke methods that return a value.

  – Environment Transition:

$$\langle l, c_i\rangle \xrightarrow{\texttt{C.}b\texttt{(args)}} \langle l, c_i'\rangle$$

  An environment transition does not access local state and always modifies the collection state, i.e., $c_i \neq c_i'$.
  An environment transition is enabled if:

  * $\exists c_o, c_o' \in \mathcal{C}$ and $\exists l', l'' \in \mathcal{L}$ s.t. the following two main transitions exist:

$$\langle l, c_i\rangle \xrightarrow{\texttt{x=C.}b_i\texttt{(args)}} \langle l', c_o\rangle$$

$$\langle l, c_i'\rangle \xrightarrow{\texttt{y=C.}b_j\texttt{(args)}} \langle l'', c_o'\rangle$$

    where r_1 is the return value of b_i's invocation, r_2 is the return value of b_j's invocation, and b_i=b_j.
  * $r_1 \neq r_2$.

Note that the environment transition `C.b(args)` influences the return value of `x=C.`b_i`(args)` (b_i = b_j) in state c_i. For example, if $c_i = \emptyset$ and `x=C.`b_i`(args)` is `x=C.get(2)` then `C.b(args)` could be a `C.put(2,4)`.

3.2 Traces and Exploration Procedure

A trace $\pi = \langle l_0, c_0\rangle \xrightarrow{s} \langle l_1, c_1\rangle \ldots \xrightarrow{s} \langle l_n, c_n\rangle$ is a sequence of transitions such that:

- $\langle l_0, c_0\rangle$ is an initial state.
- $\forall i. 0 \leq i < n. \langle l_i, c_i\rangle \xrightarrow{s} \langle l_{i+1}, c_{i+1}\rangle \in \Pi$.
- $\langle l_{n-1}, c_{n-1}\rangle \xrightarrow{\texttt{return e}} \langle l_n, c_n\rangle$ is a main transition (denoting completion of C.M).

We denote the set of traces as $[\![\Pi]\!]$.

A program trace is *sequential* if does not include environment transitions. The following provides a constructive linearizability check performed directly on traces.

DEFINITION 3.1 (LINEARIZABLE TRACE). *A trace $\langle l_0, c_0\rangle \dots \langle l_i, c_i\rangle \xrightarrow{s} \langle l_{i+1}, c_{i+1}\rangle \dots \langle l_n, c_n\rangle$ is* linearizable *iff there exists a unique i s.t. the only possible update of the collection by a main transition occurs in transition $\langle l_i, c_i\rangle \xrightarrow{s} \langle l_{i+1}, c_{i+1}\rangle$, and for every sequential run π_s starting in a state $\langle l_0, c_i\rangle$ and ending with a state $\langle l', c'\rangle$ it is the case that $c' = c_{i+1}$ and the return values of π_s and π are equal.*

We refer to $\langle l_i, c_i\rangle \xrightarrow{s} \langle l_{i+1}, c_{i+1}\rangle$ from Definition 3.1 as *linearization point*.

Checking vs. Verification.

In [26], the authors show that for the map interface, a composed operation using only `put`, `get`, and `putIfAbsent` operations is non-linearizable if and only if there exists a non-linearizable run of the presented approach. Unfortunately, while useful for checking, the procedure cannot be used to verify linearizability of an arbitrary C.M. The reason is that for verification, one needs to explore an unbounded number of inputs and unbounded number of environment operations, clearly infeasible.

In the next section, we show that verifying linearizability of a data-independent method C.M can be done by only considering a bounded number of inputs and a bounded number of collection sizes.

4. DATA-INDEPENDENT COMPOSED OPERATIONS

In this section we define data-independent classes of composed operations. The advantage of such restricted operations is that they can be verified for linearizability by considering a bounded number of inputs and a bounded number of collection entries. While verifying data-independence is undecidable in general [31], we provide simple sufficient restrictions that can establish a composed operation as data-independent.

The main idea is to restrict the inputs and the control-flow of the composed operation C.M to guarantee that the operation treats all keys uniformly. First, we limit the parameters of C.M to a key parameter and an optional value parameter. Then, we restrict the effect of these parameters on the control flow inside C.M. Since the control flow should not depend on specific input values, we also limit how the return values of basic operations are used. Finally, we preclude certain basic operations such as `replace(k,v,v')` and `remove(k,v)`, whose effect depends on `v` and the state of the collection. Later, we relax some of these restrictions to capture a wider set of composed operations.

In this section we define data-independent composed operations and a set of restrictions that guarantee data-independence. We start by defining the notion of *trace renaming*.

DEFINITION 4.1 (TRACE RENAMING). *Given a composed operation C.M with key and value parameters, a trace π of C, and values k, v, and $v' \neq v$, we define a trace $\pi_{[k,v,v']}$ obtained from π by: (i) replacing the input key for each operation in π by k; (ii) replacing the input value for the C.M operation and for each base operation of a main transition in π by v; (iii) replacing the input value for each base operation of the environment transition in π by v'; (iv) calculating the return value of operations using the specification of C ; (v) calculating the local transitions using C.M.*

4.1 Singleton Collection Method (SCM)

We start by defining the class of data-independent composed operations referred to as **S**ingleton **C**ollection **M**ethods (SCM). This is the class of operations that is most prevalent in practice.

DEFINITION 4.2 (SCM). *A composed operation C.M with parameter k and optional parameter v, is* SCM *data-independent (from now on referred to as SCM) if:*

1. *k is used as a key in all basic collection operations.*
2. *k and v are immutable: no statement in C.M can assign to k or v.*
3. *C.M can only invoke the following methods on C :* `get(`k`)`, `put(`k`,val)`, `remove(`k`)`, *and* `putIfAbsent(`k`,val)` *(here, k is the parameter k, but val need not be the v).*
4. *if a return value r of a collection operation is used in a condition, then the condition can only check the (in)equality of r to* `null` *(this also applies when the return value is assigned to other variables).*
5. *exit statements such as* `return` *and* `throw` *can only depend on the return value of collection operations.*

Restrictions 1 and 2 guarantee that the composed operation only accesses a single key in the map and in a uniform way. Restriction 3 limits the basic operations used in composed operations. The effect on the state and the return value of operations `put(`k`,val)`, `get(`k`)`, `remove(`k`)`, and `putIfAbsent(`k`,val)` does not depend on a specific value (regardless of the input value, the operation behaves identically). Note that `putIfAbsent(`k`,`v`)` depends only on the existence of k in the map and does not depend on a specific value. The challenge is to guarantee that these operations are composed in a way that preserves data-independence. This motivates restrictions 4 and 5 limit control-flow decisions to those that depend on the existence of a value in the collection without referring to a specific value.

Examples.

Table 1 shows a composed operation, named `compute`, and a composed operation named `Replace`, together with their update and return value. Both the update and return value of `compute` do not depend on a specific value. However, in `Replace`, both the update and return value depend on the input argument v'. Indeed, `compute` satisfies the restrictions from Definition 4.2, while `Replace` does not satisfy Restriction 4 due to condition `get(k) == v'` inside the `if` statement.

Consider again the `getLock` operation in our running example in Figure 2. This composed operation meets all of the conditions of Definition 4.2 and is therefore an SCM. The operation uses the underlying `ConcurrentHashMap locks` to memoize the `ReentrantLock` allocated at line 4. When the value for a given `id` is cached in the collection it is returned immediately; when the `ReentrantLock` for a given `id` is not available, it is allocated and inserted into the collection. . It has a single input `id` which is used only as a key in the `ConcurrentHashMap locks`. And the return value of the collection operation is only checked for (in)equality to `null` (lines 3 and 7).

Next, based on the restrictions from Definition 4.2, the map specification, and the non-commutativity specification of maps, we state the main reduction theorem. We use the simple case of SCM where k is used as a key and v is used as a value in all operations. Theories for composed operations satisfying other restrictions can be stated and proved in a similar way. Due to space limitation we omit the proofs from the paper. The full details can be found in [26].

THEOREM 4.1. *If C is a collection where C.M is an SCM operation, and π is a linearizable trace of C, then for any input key k and values $v, v' \neq v$, $\pi_{[k,v,v']}$ is a linearizable trace of C.*

Table 1: **Sample composed operations and their specifications.** `compute` **is an SCM operation and** `Replace` **is not an SCM operation.** $M : \mathbb{N} \to \mathbb{N}$ **denotes the content of the map.**

Operation	Updated Map Value	Return Value
$compute(k, v)\{$ $nv = get(k);$ $if(nv \neq null)\{$ $nv \ = \ putIfAbsent(k, v);$ $if(nv = \texttt{null})$ $nv = v;$ $\}$ $return\ nv;$ $\}$	$\begin{cases} M & M(k) \neq null \\ M[k \mapsto v] & o/w \end{cases}$	$\begin{cases} M(k) & M(k) \neq null \\ v & o/w \end{cases}$
$Replace(k, v, v')\{$ $if(get(k) == v')\{$ $return\ replace(k, v, v');$ $return false;$ $\}$	$\begin{cases} M & M(k) \neq v' \\ M[k \mapsto v] & o/w \end{cases}$	$\begin{cases} false & M(k) \neq v' \\ true & o/w \end{cases}$

```
1 V doubleNonLin(K) {
2   val = m.get(K);
3   if (val == null) {
4    nv = K*2;
5    m.putIfAbsent(K,nv);
6   }
7   val = m.get(K);
8   return val;
9 }
```

(a)

```
1 V doubleLin(K) {
2   val = m.get(K);
3   if (val == null) {
4    nv = K*2;
5    val=m.putIfAbsent(K,nv);
6    if (val == null)
7       val = nv;
8   }
9   return val;
10 }
```

(b)

Figure 3: **(a) an example of a non-linearizable method inspired by bugs from** `Adobe BlazeDS`**,** `Vo Urp` **and** `Ehcache-spring-annotations`**. (b) a possible linearizable fix for the example from (a). Both examples are SCMs.**

The theorem combined with the map semantics imply the following corollary.

COROLLARY 4.2. *Checking linearizability of SCM operations can be done using a fixed set of map entries.*

Note that for programs in which these are the only infinite values, the problem becomes decidable.

Also note that Theorem 4.1 shows that verification of linearizability of SCM composed operations can be done using any key and any value. Therefore, verifying linearizability of SCM operations can be done by exploring all traces generated by the transition relation from Section 3 using a single input key and a single input value and checking whether the specification from Definition 3.1 holds for these traces. This can be accomplished using a standard bounded model checker such as SPIN.

4.2 Example

We now show a non-linearizable example, together with a trace generated using our algorithm, showing a linearizability violation. We also show a possible fix to the example and demonstrate the runs generated by our algorithm on the fixed linearizable example.

Figure 3a shows a non-linearizable method `doubleNonLin` inspired by bugs from `Ehcache-spring-annotations`, `BlazeDS`, and `Vo Urp`. The method uses an underlying concurrent collection `m` to memoize the value `K*2` for each `K`. When the value for a given key is cached in the collection it is read and returned; when the value for a given key is not available, it is computed and inserted into the collection and then read and returned.

Figure 4a shows a trace exposing a linearizability violation of `doubleNonLin` that was generated using our algorithm. In this figure, a state is represented by a box, a linearization point is represented by a black box (recall that the term linearization point was defined earlier: a transition in the execution where the operation takes effect), a state of the sequential run is represented by a circle, arrow represents a transition, and a dashed arrow represents an environment transition together with its operation.

Our algorithm generated the trace as follows: The run starts at the initial state $\langle pc = 1, K = 1, C = \emptyset \rangle$ and runs sequentially until state $\langle pc = 7, C = \{(1, 2)\} \rangle$. This state is a linearization point, therefore, at this point, a sequential run starts from the state $\langle pc = 1, K = 1, C = \emptyset \rangle$. While $pc = 1, K = 1$ is the local initial state of `doubleNonLin` (K is the input argument), $C = \emptyset$ is copied from the state previous to the linearization point. The sequential run continues without intervening environment operations until termination (sequential trace represented by circles). Upon termination, the map entries at the sequential trace's last state and at the linearization point's state are compared and in this case are equivalent (otherwise, a linearization violation was found). At this point, the run continues and an environment operation, `m.put(1,3)`, that influences the next to come operation `m.get(3)` is performed. Then, the run continues until the return statement. In this trace, the linearizability violation is revealed due to a different return value at the concurrent trace's last state($val = 3$) and at the last sequential trace's state($val = 2$).

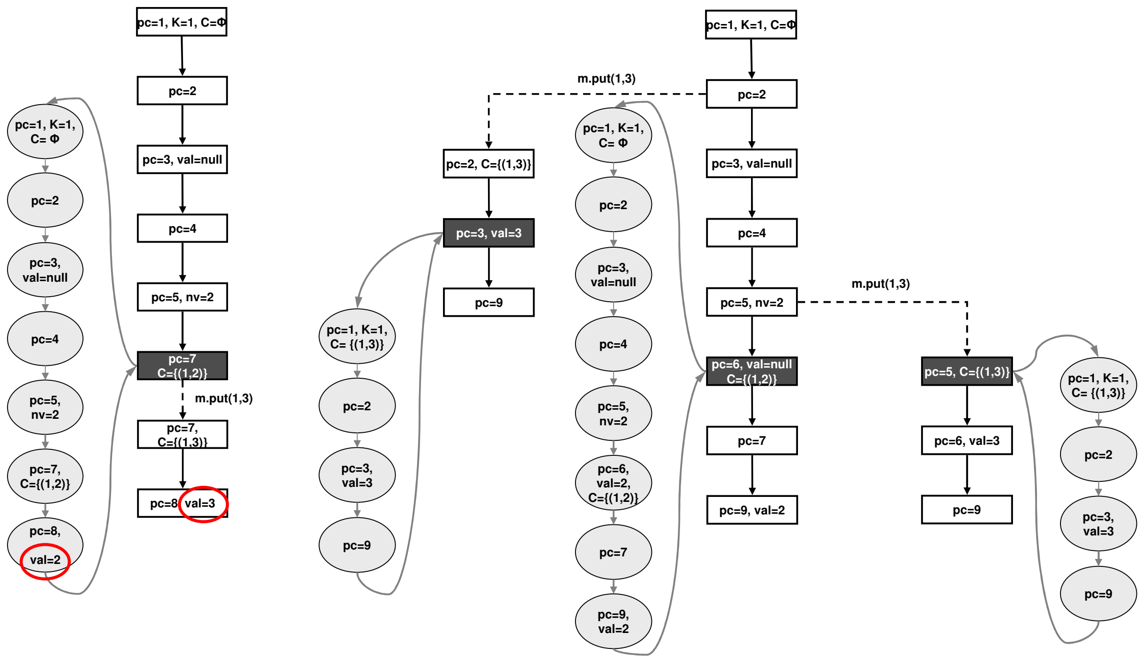
Method `doubleLin`, presented at Figure 3b, is a corrected version of the method `doubleNonLin` from Figure 3a. Figure 4b shows all the runs generated by our algorithm on `doubleLin`. Because `doubleLin` is SCM, we used only a single input to verify the linearizability of the custom collection built by `doubleLin`. In addition, we restricted the environment to write a single value other than the one calculated by `doubleLin`, in this case the value 3 was used. It is easy to see that in each trace the return value is the same as the return value returned by the sequential trace. Therefore, since all runs generated by our algorithm using a single input key meet the condition of Definition 3.1, we conclude that `doubleLin` is linearizable.

4.3 Value Collection Method (VCM)

We generalize the definition of SCM to deal with composed operations containing the operation `replace(k,v,v')`. We refer to this class as **V**alue **C**ollection **M**ethods (VCM). In VCM, the operation `replace(k,v',v)` is allowed to appear in the composed operation as long as `v'` is a value returned by a `get(k)` operation.

Figure 5 shows an example of a composed operation `inc` taken from OpenJDK [9]. This operation implements an increment action for a concurrent histogram (as a map of counters). This operation is not an SCM because it has a `replace(k,v',v)` operation at line 8. However, this composed operation is VCM because the value `i` used as `v'` in line 8 is extracted from the collection at line 3.

Note that Theorem 4.1 can be generalized to deal with VCM operations which therefore can be verified as linearizable using any



(a) (b)

Figure 4: Runs generated by our algorithm. A state is represented by a box, a linearization point is represented by a black box, A state of the sequential run is represented by a circle, arrow represents a transition, and a dashed arrow represents an environment transition together with its operation. (a) shows a run, generated by our algorithm, of `doubleNonLin` method from Figure 3a showing a linearizability violation. The violation revealed by the different return value (`val`) at the end of the sequential trace. (b) shows all the traces, generated by our algorithm, for `doubleLin` method from Figure 3b.

```
void inc(Class<?> key) {
 for (;;) {
  Integer i = get(key);
  if (i == null) {
   if (putIfAbsent(key, 1) == null)
   return;
  } else {
   if (replace(key, i, i + 1))
   return;
  }
 }
}
```

Figure 5: VCM example, taken from OpenJDK [9]

key and any value. Verifying linearizability of VCM operations, as SCM, can be done by exploring all traces generated by the transition relation from Section 3 using a single input key and a single input value and checking whether the specification from Definition 3.1 holds for these traces.

4.4 Fixed Collection Method (FCM)

We also generalize the class of SCM operations to deal with programs that are dependent on a finite number of values. We refer to this more general class as **F**ixed **C**ollection **M**ethods (FCM). We generalize SCM by weakening the restriction that exit statements and collection operations are not control-dependent on expressions using k. In FCM, exit statements and collection operations may be control-dependent on expressions using k when the expression compares k to a constant value. We use these expressions to build a set of inputs I for the FCM by adding all constants, as well as, one additional value that is not in I.

```
AndroidTools forOsFamily(String osFamily) {
 AndroidTools instance = androidTools.get(osFamily);

 if(instance == null) {
   AndroidTools newInstance = null;
   if(osFamily.equals("windows")) {
     newInstance = new WindowsAndroidTools();
   }   else if (osFamily.equals("unix")) {
     newInstance = new UnixAndroidTools();
   }   else {
     throw new
      UnsupportedOperationException("...");
   }

   instance =
     androidTools.putIfAbsent(osFamily, newInstance);
   if (instance == null)
     instance = newInstance;
 }

  return instance;
}
```

Figure 6: FCM example, taken from AutoAndroid [5]

```
 1 FxValueRenderer getInstance(FxLanguage language) {
 2   if (language == null) {
 3     return renderers.get(DEFAULT);
 4   }
 5   if (!renderers.containsKey(language)) {
 6     renderers.putIfAbsent(language,
 7         new FxValueRendererImpl(language));
 8   }
 9   return renderers.get(language);
10 }
```

Figure 7: A data-dependent method example, taken from fleXive [7]

Figure 6 shows a composed operation `forOsFamily` taken from Autoandroid [5]. This operation returns an instance of AndroidTools suitable for the given operating system. This operation either returns a new instance and updates the `ConcurrentHashMap androidTools` or returns an existing instance extracted from the map. AutoAndroid supports only tools for the "unix" and "windows" operating systems, therefore, lines 6 – 13, check whether the input string is either "unix" or "windows" and throws an exception otherwise.

Clearly, this function is not an SCM because there is an abnormal exit statement (`throw` at line 11) that is control dependent on the input key `osFamily` (lines 6 and 8). However, the input key `osFamily` is only compared to the constants "`unix`" and "`windows`". Moreover, `forOsFamily` meets all the other SCM requirements. Therefore, `osFamily` is an FCM and the fixed input set is the set { "unix", "windows", X} s.t. X is any value other than "unix" and "windows".

Note that Theorem 4.1 can also be generalized to deal with FCM operations. Unlike SCM and VCM, FCM operations are dependent only on a fixed set of inputs. Therefore, they can be verified as linearizable by exploring all traces generated by the transition relation from Section 3 using this fixed set of inputs. Verifying `forOsFamily` for example, can be done using the set of input {"unix", "windows", "*"}, where "*" can be replaced with any string different that "unix" and "windows".

4.5 Example of a Data-Dependent Composed Operation

Figure 7 shows part of a composed operation `getInstance` taken from fleXive [7]. This opeartion takes an input `language` and when the input is `null` returns a predefined `FxValueRenderer` corresponding to the key `DEFAULT` in the `ConcurrentHashMap renderers` (lines 2 – 3). Otherwise, the method either returns a new object `FxValueRenderer` and updates the `renderers` or returns an existing `FxValueRenderer` from `renderers` (lines 5 – 9).

This composed operation is not linearizable, but can be repaired to be linearizable, as shown in Figure 8. However, even its linearizable version is neither SCM nor FCM. The reason is that in line 3 there is an access to `renderers` with the key `DEFAULT` which might be other that the value of `language`.

5. EXPERIMENTAL EVALUATION

In this section we outline the implementation of our tool and evaluate its effectiveness for checking linearizability of composed operation on a wide range of real-world applications. Using our approach, we show that all linearizable real-world composed operations we identified can be categorized as one of SCM, VCM, or FCM and can be handled by our technique. We also classify those operations which do not fall in these classes (we refer to those operations as data-dependent).

```
 1 FxValueRenderer getInstance(FxLanguage language) {
 2  if (language == null) {
 3    return renderers.get(DEFAULT);
 4  }

 6  FxValueRenderer oldV = renderers.get(language);

 8  if (oldV == null) {
 9   FxValueRenderer V = new FxValueRendererImpl(language);
10   oldV = renderers.putIfAbsent(language, V);

12   if (oldV == null)
13      oldV = V;
14  }
15  return oldV;
16 }
```

Figure 8: The data-dependent `getInstance` method, taken from fleXive [7], fixed to be linearizable.

5.1 Implementation

An outline of the tool implementation is shown in Figure 9. The programmer provides a multithreaded program to the `Composed Operation Extractor` (referred to as `CO Extractor`). The `CO Extractor` uses the WALA [10] static analyzer to identify composed operations which include multiple base collection operations.

The extracted composed operation (CO) is then provided to the VERIFIER module. This module checks whether the given composed operation is SCM, FCM or VCM. If it is either of these three, then it is considered to be data-independent, otherwise the operation is considered data-dependent. If it is an FCM, the VERIFIER module also provides a set of inputs. Further, to simplify checking, our tool takes as input additional specification in the form of (potentially conditional) linearization points of the composed operation. When VERIFIER returns a data-dependent result the composed operation is returned to the user since it cannot be handled by our approach.

When the VERIFIER returns an SCM, an FCM, or a VCM, we automatically generate Promela models, which are fed to the SPIN model checker in order to verify linearizability. This can be done for composed operations with primitive keys and values. However, for composed operations with more complex keys or values (e.g. of type String), we require that the programmer provides an input driver for generating the actual values for keys and values. In addition, in this case, the user provides an influence driver that gets an input object and returns a single different object. This driver is used by the environment of our technique.

Note that for SCM, the input driver needs to provide only an arbitrary key and an arbitrary value and the influence driver needs to provide another value which differs from the one provided by the input driver. Also note that the sequential specification we used for checking linearizability is by executing the custom collection operations in an atomic manner.

5.2 Applications

Table 2 lists 57 real-world applications using Java's concurrent collections. In many applications, concurrent collections were introduced to address observed scalability problems, replacing manual locking of a sequential map. Each of the applications contains at least one method that was extracted and tested by our tool. In 33 out of the 112 extracted methods, the `CO Extractor` identified a composed operation inside a large method and we manually extracted and generated the composed operation. The extracted methods together with explanations for each method can be found at [6].

Table 2: 57 **applications used for experiments**

Program	LOC	Description
Adaptive Planning	1,103,453	Automated budgeting tool
Adobe BlazeDS	180,822	Server-based Java remoting
Amf-serializer	4,553	AMF3 messages serializaton
Annsor	1,430	runtime annotation processor
Apache Cassandra	54,470	Distributed Database
Apache Derby	618,352	Relational database
Apache MyFaces	201,130	JSF framework
Apache ServiceMix	78,340	Enterprise Service Bus
Apache Struts	110,710	Java web apps framework
Apache Tomcat	165,266	Java Servlet
Apache Wicket	142,968	Web application framework
ApacheCXF	311,285	Services Framework
Autoandroid	19,764	Tools for automating android projects
Beanlib	42,693	Java Bean library
Carbonado	53,455	Java abstraction layer
CBB	16,934	Concurrent Building Blocks
Clojure	25,421	dynamic programming language for the JVM
cometdim	5,571	A web IM project
Daisy	334,337	Content and information management
DWR	26,094	Ajax for Java
dyuproject	26,593	Java REST framework
Ehcache Annotations for Spring	3,184	Automatic integration of Ehcache in spring projects
Ektorp	6,261	Java API for CouchDB
EntityFS	79,820	OO file system API
eXo	13,298	Portal
FindBugs	106,031	Static analysis tool
fleXive	910,780	Java EE 5 content repository
GlassFish	260,461	JavaServer faces
Granite	28,932	Data services
gridkit	8,746	Kit of data grid tool and libs
GWTEventService	17,113	Remote event listening for GWT
Hazelcast	59,139	Data grid for Java
Hudson	14,991	Automatic build system
hwajjin	4,371	A Struts plugin for Java
ifw2	54,888	Web application framework
Jack4j	4,477	Interface to Jack library
JBoss AOP	1,013,073	Aspect oriented framework
Jetty	64,039	Java HTTP servlet server
Jexin	11,024	functional testing platform
JRipples	148,473	Program analysis
JSefa	27,208	Object serialization library
keyczar	4,720	Cryptography Toolkit
memcache-client	4,884	Memcache client for Java
Module Glassfish	745	Module for Glassfish
OpenEJB	191,918	Server
OpenJDK	1,634,818	JDK 7
P-GRADE	1,154,884	P-GRADE Grid Portal
Project Tammi	163,913	Java development framework
Project Track	5,160	Example application
RESTEasy	81,586	Java REST framework
Retrotranslator	27,315	Automatic compatibility tool
Streamy	483,418	Audio/video recorder
Tersus	165,160	Visual Programming Platform
torque-spring	2,526	Torque support classes
Vo Urp	24,996	UML data models translator
WebMill	57,161	CMS portal
Xbird	196,893	XQuery processor
Yasca	326,502	Program analysis tool

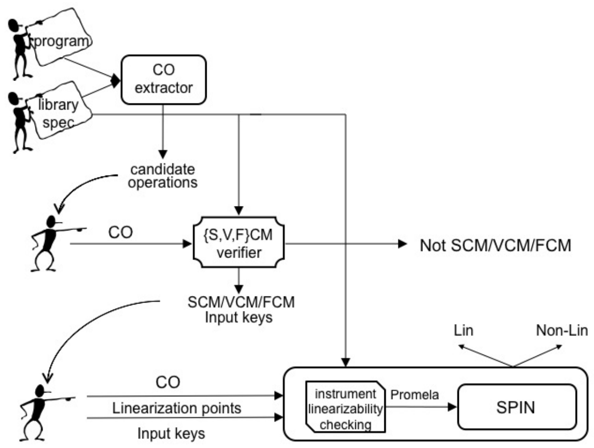


Figure 9: **Technique overview**

5.3 Results

We tested 112 methods in 57 applications. All of our experiments were carried out using an AMD Opteron 2.4Ghz dual hyper threaded CPUs, 8GB RAM platform running on a 64 bit Linux. The runtime for analyzing each method was less than a second.

Figure 10 (a) shows that 49 methods are data-independent. Out of these, 43 are SCM, 1 is VCM, and 5 depend on fixed input keys (FCM). Furthermore, 30 methods were verified as linearizable, and a linearizability violation was detected in the other 19 methods. We then manually fixed the linearizability violations in all of these 19 methods and verified their fixed versions are linearizable.

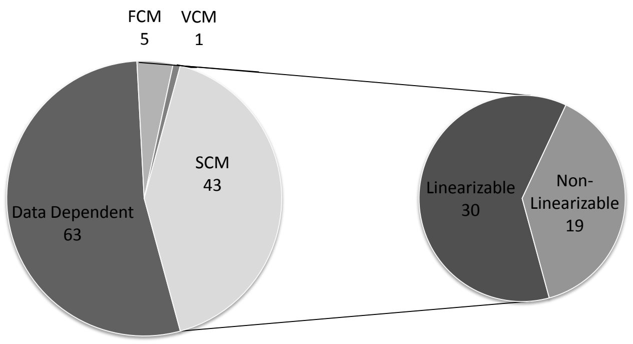
Figure 10 (a) also shows that 63(56%) of the tested methods are data-dependent. The data-dependent methods cannot be verified by our technique. However, a close inspection revealed that all the data-dependent methods are non-linearizable. Moreover, it is not obvious if these methods can be rewritten as equivalent linearizable methods because, as Section 5.3.1 shows, the results of most of these methods depend on the values of global variables. In summary, all of the linearizable methods were verified with our technique.

Figure 10 (b) shows the distribution of the data-independent methods that are linearizable, or can be fixed to be linearizable in each of our benchmarks. In 21 out the 57 application we automatically verified the linearizability of all composed operations. The remaining 36 applications also include data-dependent methods that cannot be verified by our technique.

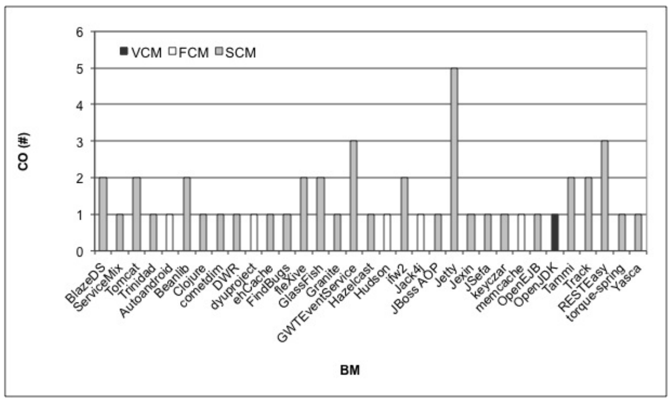
5.3.1 Benchmark Classification

Figure 11 shows the classification of our 63 data-dependent composed operations according to the rules they falsify. The largest group, marked “Globals”, has 43 operations that use global variables and are therefore not encapsulated. These operations are non-linearizable in an open environment. Moreover, it is not obvious whether these methods can be rewritten as equivalent linearizable methods because the result of these methods depends on the values of the globals. For example, Figure 12 shows a method from the Granite benchmark, where the result depends on the global variables `_proxy`, and `_noproxy`(note the condition in line 10). An environment that changes the value of `_proxy` affects the value of `destination`, calculated in line 12.

In 2 of the data-dependent operations there is a `remove(k,v)` operation (marked as “Remove”), 6 may exit not based on collection result (marked as “Exit”), and 3 access the map with two different keys (marked as “Keys”). An example of the last case



(a)



(b)

Figure 10: **Benchmark results. (a) classification of** 112 **benchmark methods into SCM, VCM, FCM, and data dependent methods, and classification of the** 49 **data-independent methods into linearizable and non-linearizable. The rest** 63 **data-independent methods are non-linearizable (b) data-independent methods that are either linearizable or can be fixed to be linearizable in each one of our benchmarks.**

```
public HttpDestination getDestination(Address remote, boolean ssl) throws UnknownHostException, IOException
{
   if (remote == null)
     throw new UnknownHostException("Remote socket address cannot be null.");

   HttpDestination destination = _destinations.get(remote);
   if (destination == null)
   {
     destination = new HttpDestination(this, remote, ssl, _maxConnectionsPerAddress);
     if (_proxy != null && (_noProxy == null || !_noProxy.contains(remote.getHost())))
     {
        destination.setProxy(_proxy);
        if (_proxyAuthentication != null)
          destination.setProxyAuthentication(_proxyAuthentication);
     }
        HttpDestination other =_destinations.putIfAbsent(remote, destination);
        if (other!=null) destination=other;
   }
   return destination;
}
```

Figure 12: **A non-linearizable composed operation from GRANITE. The result of the operation depends on the global variables `_proxy` and `_noProxy`.**

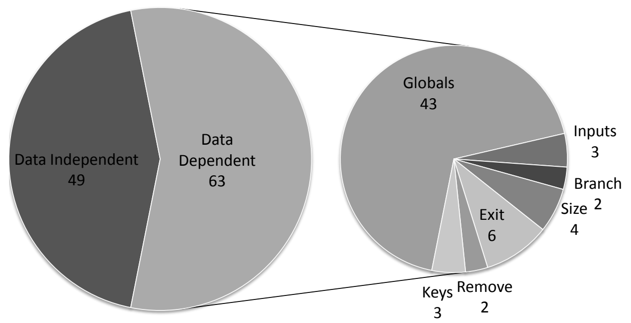


Figure 11: **Classification of our** 112 **data-dependent and data-independent composed operations according to the rules they falsify or satisfy.**

is the function `getInstance` shown in Figure 7. This method accesses the `DEFAULT` key in line 3 and the `language` key at lines 6 and 10. Finally, in 2 of the data-dependent composed operations there exists a branch on collection return value other than `null` ("Branch"), 4 have a `size` ("Size"), and in 3 the composed operation has more than two input arguments ("Inputs").

6. RELATED WORK

Next, we survey some of the work that is most closely related to our approach.

6.1 Dynamic Analysis of Linearizabilty

Dynamic atomicity checkers such as [20, 19] check for violations of conflict serializability. As noted earlier, conflict-serializability is inappropriate in our setting as typically the underlying base collection is linearizable and programmers simply want to add other linearizable methods. Hence, we focus on checking linearizability.

Linearizability checking tools have proven quite effective in finding bugs. Line-Up [14] is a dynamic linearizability checker that learns the sequential specification dynamically and then checks it for linearizability. [27] is a dynamic tool for checking linearizability of composed collection operations.

All of the above techniques can effectively find bugs in general programs, however, these methods may produce false negatives. In this paper, we focus on data-independent programs manipulating maps and present a procedure that can *prove* linearizability for that class.

6.2 Linearizability Verification

A manual proof of correctness of several interesting concurrent data structure implementations using rely-guarantee reasoning is presented in [29].

The PVS system has been successfully used to semi-automatically verify linearizability [18, 16, 21] of several programs. These proofs provide crucial insights on essential points of the algorithm. However, this is a very time consuming task that needs to be repeated for each new program. In contrast, our approach works on a restricted class of programs but is much more automatic.

The work of [12] introduced the idea of using abstract interpretation [17] to develop an automatic over-approximation for checking linearizability. Thus, the algorithm can prove linearizability in certain programs but may fail due to overly conservative abstraction. The algorithm assumes that the concurrent and sequential runs differ by at most one element, thereby drastically simplifying the task of checking linearizability. The work in [13, 25] generalizes [12] using a thread-centric approach to verify programs with an unbounded number of threads. In [28], the idea of bounded difference is combined with rely guarantee reasoning and shape abstractions to perform fast linearizability checks. In contrast, the work in this paper is not only sound but also guaranteed to be complete for data independent collection manipulations. Note that we prove that linearizability violations can be observed using maps with at most one key which implies bounded differences.

Works such as [30] focus on model checking individual collections. Our work operates at a higher level, as a client of already verified collections, and leverages the specifications of the underlying collections and the composed operation's data-independent characteristics to reduce the search space. Also, we bound the global state without using any approximation.

In [15] is shown that checking linearizability is decidable for programs with simple linked-list manipulations. We focus on collections that hide the internal representation and use data-independence to show that the state space reduces to the state space of the local state. This implies decidability for finite-state local states and is effective for all the data-independent programs we have encountered.

6.3 Effective Techniques for State Space Reduction

This paper uses data-independence which was inspired by [31] to bound the size of the global state space.

Partial order reduction techniques, such as [22], utilize commutativity of individual memory operations to filter out execution paths which cannot lead to new violations. While this technique has proven to be very effective for drastically reducing the state space in explicit state model checking, it is in general difficult to infer commutativity for real life software. This paper uses the fact that the abstract interface of the collection is known at library design time to build a tool which incorporates commutativity checking.

7. CONCLUSION AND FUTURE WORK

Proliferation of concurrent libraries combined with desire for increased performance has led to the current situation where programmers build applications that create custom composed concurrent operations. These composed concurrent operations consist of multiple invocations of underlying library operations and are meant to be linearizable. Unfortunately, implementing such composed concurrent operations correctly and efficiently has proven to be quite difficult.

In this paper, we identified a class of data-independent composed concurrent operations that frequently arises in practice. Then, we presented a novel automatic linearizability verification procedure for that class. The basic insight is to leverage the restricted structure of the composed concurrent operations by defining a reduction procedure which enables us to verify the composed operation by only considering a small *bounded* number of values. We implemented the reduction in a tool and used it to prove linearizability of practical composed collections which were automatically extracted from a large variety of open source projects.

As future work, we plan to extend the class of data-independent composed operations as well as to study their use in other programming languages.

8. ACKNOWLEDGMENTS

This work was supported in part by NSF grant CCF-1160904.

9. REFERENCES

[1] http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166edocs/jsr166e/ConcurrentHashMapV8.html.

[2] http://old.nabble.com/ConcurrentHashMapV8-td32352891i20.html.

[3] Apache servicemix. http://servicemix.apache.org/.

[4] Apache tomcat. http://tomcat.apache.org/.

[5] autoandroid. http://code.google.com/p/autoandroid.

[6] Benchmark programs. https://docs.google.com/document/pub?id=1ik0MQKebMrupYjE8yeWwF7PxaYgO1GmL5P07SjpIxXo.

[7] flexive. https://sourceforge.net/projects/flexive/.

[8] Koders. http://www.koders.com/.

[9] openjdk. http://hg.openjdk.java.net/jdk7/jaxp/jdk.

[10] T.j. watson libraries for analysis (wala). http://wala.sourceforge.net/wiki/index.php/Main_Page.

[11] ALUR, R., MCMILLAN, K., AND PELED, D. Model-checking of correctness conditions for concurrent objects. In *LICS* (1996), p. 219.

[12] AMIT, D., RINETZKY, N., REPS, T. W., SAGIV, M., AND YAHAV, E. Comparison under abstraction for verifying linearizability. In *CAV* (2007), pp. 477–490.

[13] BERDINE, J., LEV-AMI, T., MANEVICH, R., RAMALINGAM, G., AND SAGIV, M. Thread quantification for concurrent shape analysis. In *CAV* (2008).

[14] BURCKHARDT, S., DERN, C., MUSUVATHI, M., AND TAN, R. Line-up: a complete and automatic linearizability checker. In *PLDI* (2010).

[15] CERNY, P., RADHAKRISHNA, A., ZUFFEREY, D., CHAUDHURI, S., AND ALUR, R. Model checking of linearizability of concurrent list implementations. In *CAV*. 2010, pp. 465–479.

[16] COLVIN, R., GROVES, L., LUCHANGCO, V., AND MOIR, M. Formal verification of a lazy concurrent list-based set algorithm. In *CAV* (2006).

[17] COUSOT, P., AND COUSOT, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL* (1977), pp. 238–252.

[18] DOHERTY, S., GROVES, L., LUCHANGCO, V., AND MOIR, M. Formal verification of a practical lock-free queue algorithm. In *FORTE* (2004).

[19] FLANAGAN, C., AND FREUND, S. N. Atomizer: A dynamic atomicity checker for multithreaded programs. In *POPL* (2004), pp. 256–267.

[20] FLANAGAN, C., FREUND, S. N., AND YI, J. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI* (2008), pp. 293–303.

[21] GAO, H., AND HESSELINK, W. H. A formal reduction for lock-free parallel algorithms. In *CAV* (2004).

[22] GODEFROID, P. Partial-order methods for the verification of concurrent systems - an approach to the state-explosion problem, 1995.

[23] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. *TOPLAS 12*, 3 (1990).

[24] HOLZMANN, G. J. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.

[25] MANEVICH, R., LEV-AMI, T., SAGIV, M., RAMALINGAM, G., AND BERDINE, J. Heap decomposition for concurrent shape analysis. In *SAS* (2008), pp. 363–377.

[26] SHACHAM, O. *Verifying Atomicity of Composed Concurrent Operations*. PhD thesis, Tel Aviv University, 2012.

[27] SHACHAM, O., BRONSON, N., AIKEN, A., SAGIV, M., VECHEV, M., AND YAHAV, E. Testing atomicity of composed concurrent operations. In *OOPSLA* (2011), pp. 51–64.

[28] VAFEIADIS, V. Automatically proving linearizability. In *CAV*. 2010.

[29] VAFEIADIS, V., HERLIHY, M., HOARE, T., AND SHAPIRO, M. Proving correctness of highly-concurrent linearisable objects. In *PPoPP* (2006).

[30] VECHEV, M., YAHAV, E., AND YORSH, G. Experience with model checking linearizability. In *SPIN* (2009), pp. 261–278.

[31] WOLPER, P. Expressing interesting properties of programs in propositional temporal logic. In *POPL* (1986).