Automatic Atomicity Verification for Clients of Concurrent Data Structures

Mohsen Lesani, Todd Millstein, and Jens Palsberg

University of California, Los Angeles
{lesani,todd,palsberg}@cs.ucla.edu

Abstract. Mainstream programming languages offer libraries of concurrent data structures. Each method call on a concurrent data structure appears to take effect atomically. However, clients of such data structures often require stronger guarantees. For instance, a histogram class that is implemented using a concurrent map may require a method to atomically increment a histogram bar, but its implementation requires multiple calls to the map and hence is not atomic by default. Indeed, prior work has shown that atomicity errors in clients of concurrent data structures occur frequently in production code.

We present an automatic and modular verification technique for clients of concurrent data structures. We define a novel sufficient condition for atomicity of clients called condensability. We present a tool called Snowflake that generates proof obligations for condensability of Java client methods and discharges them using an off-the-shelf SMT solver. We applied Snowflake to an existing suite of client methods from several open-source applications. It successfully verified 76.9% of the atomic methods without any change and verified the rest of them with small code refactoring and/or annotations.

1 Introduction

Many modern programming languages provide libraries of *concurrent data structures* (e.g., the `java.util.concurrent` package and Intel Threading Building Blocks library) that are widely used. A concurrent data structure is an object that satisfies the well-known correctness criterion called *linearizability* [19]. At a high level, this property ensures that the operations of the data structure can be invoked concurrently from multiple threads while still appearing to execute atomically and behaving according to the sequential specification of the data structure. The linearizability guarantee relieves the programmer from complex reasoning about possible interference among data-structure methods and removes the need to add explicit synchronization.

While the linearizability guarantee is very useful, it only pertains to an *individual* operation on the data structure. In practice, clients of a concurrent data structure may require stronger guarantees. For example, consider the `AtomicMap` class in Figure 1, which is a subset of Java's `ConcurrentHashMap` class and provides atomic methods for getting, putting and removing elements, as well as

A. Biere and R. Bloem (Eds.): CAV 2014, LNCS 8559, pp. 550–567, 2014.

```
class AtomicMap<K, V> { // data structure
  V get(K k) { /*..*/ }
  void put(K k, V v) { /*..*/ }
  V remove(K k) { /*..*/ }
  V putIfAbsent(K k, V v) { /*..*/ }
  boolean replace(K k, V ov, V nv) { /*..*/ }
}
```

```
class AtomicHistogram<K> { // client
  private AtomicMap<K, Integer> m;

  V get(K k) {
    return m.get(k);
  }

  Integer inc(K key) {
    while (true) {
      Integer i = m.get(key);
      if (i == null) {
        Integer r = m.putIfAbsent(key, 1); ⊛
        if (r == null)
          return 1;
      } else {
        Integer ni = i + 1;
        boolean b = m.replace(key, i, ni); ⊛
        if (b)
          return ni;
} } } }
```

Fig. 1. The classes `AtomicMap` and `AtomicHistogram`

conditional versions of `put`: `putIfAbsent` only performs the `put` if the given key is currently unmapped, and `replace` only performs the `put` if the given key is currently mapped to a given value. As Figure 1 shows, a programmer may use the `AtomicMap` class to implement the client `AtomicHistogram` class, which supports the method `inc` to increment one bar of the histogram. The figure shows a correct implementation of atomic increment [30], which is subtle and error prone. For example, a naive implementation of this client method, which simply `get`s the current value and `put`s back an incremented value, is not atomic and can easily violate the sequential specification in the presence of multiple threads. In this paper, we present an automatic and modular technique for verification of the atomicity of clients of concurrent data structures, such as our histogram class.

Prior work on automatic atomicity verification leverages Lipton's notion of *moverness* [23]. Moverness can be applied to verify conflict-serializability of transactions [4] and atomicity of both data-structure and client methods [14, 15, 35].

The main idea is to prove that individual operations in a method M can commute with operations from other threads, in such a way that M's operations can be always "moved" to be contiguous in any execution. Moverness has been successfully applied to automatically check atomicity of concurrent code that uses locks for synchronization [14, 15] and was later extended to support non-blocking synchronization by paired load-link (LL) and store-conditional (SC) instructions [35]. Unfortunately, the ABA problem [27] makes moverness too strong a requirement to prove atomicity of non-blocking algorithms that employ compare-and-swap (CAS) [35]. Similarly, as we will show in the next section, the ABA problem makes the moverness requirement too strong to prove the atomicity of the increment method in Figure 1.

Instead, we define and check a novel sufficient condition for atomicity called *condensability*. Our approach handles client classes that use a single concurrent data structure in their implementation. Consider a client method M that uses an atomic object o. Intuitively, a call to M in a concurrent execution e is condensable if there is a method call m on o in M's execution such that (a) either m does not modify the state of o or it is the only method call in M's execution that does so; and (b) the sequential execution of the entire method M at the place of m in e results in the same final state of o as m and the same return value as the original execution of M. A client object is condensable if every execution of every method of it is condensable. The notion of condensability is similar in spirit to the idea of moverness, but instead of moving individual operations in a method, condensability allows relocating the entire method at once. Condensability targets a common class of clients that access a single concurrent data structure and provides a modular verification technique for atomicity of this class of clients. Specifically, condensability can be separately checked for each method, so changes to one method do not affect the condensability of other methods. In Section 3, we formalize condensability and prove that condensability implies atomicity.

We demonstrate the applicability of condensability with an automatic checking tool for Java called Snowflake. The tool takes as input a client class C along with a sequential specification for each of the methods in the concurrent data structure that C employs. As we will show later, such specifications are typically quite simple and are obtainable from documentation of the data structures. For each method in C, Snowflake generates a set of proof obligations that are sufficient for condensability and provides them to the Z3 SMT solver [8]. If the proof obligations are discharged, the method is verified to be atomic.

We applied Snowflake to a suite of open-source benchmarks that was used to evaluate prior work by others [30]. Snowflake succeeds in verifying atomicity of 76.9% of the atomic methods and rejecting all non-atomic methods in the benchmark suite. In addition, Snowflake can verify the remaining 23.1% of the atomic methods after some manual code refactoring.

Related Work. Shacham et al. [30] provide a tool called Colt for finding atomicity bugs in client methods of concurrent data structures by heuristically executing such code with interference from other threads. They reported many bugs

in a variety of real-world applications. Tools like Colt identify actual executions with atomicity bugs and as such have no false positives, but they cannot prove the absence of such errors.

In later work, Shacham and colleagues have explored conditions on client methods that allow for exhaustive testing for interference, thereby supporting atomicity verification. Shacham [29] shows that a *data-independent* client method, whose control flow does not depend on the specific data values used, need only be tested using a bounded number of data values in order to cover all possible atomicity violations. Zomer et al. [37] show that an *encapsulated* client method, whose only shared state is the underlying data structure, need only be tested using two threads and one occurrence of the client method. They also provide a condition called *composition closure* on the underlying data structure that allows each client method to be tested separately for interference. Our work requires client methods to be encapsulated and to support additional restrictions but does not restrict the data structure itself; indeed maps are not composition closed. Our restrictions allow us to verify atomicity via a few simple and modular condensability conditions on each method.

Work on atomicity refinement provides sound rules for extending the scope of atomic blocks [12, 20]. Some refinement rules, such as Jonsson's *absorption rule* [20], are similar in spirit to our requirements for condensability. However, the refinement rules must be applied step by step in order to eventually produce a single atomic block, while condensability directly compares an interleaved execution to a sequential version.

Others have ensured atomicity for clients of linearizable data structures by automatically inserting additional synchronization [5, 16, 18]. Such approaches provide strong atomicity guarantees by construction but incur synchronization overheads that our approach avoids.

In addition to prior work on atomicity, condensability is closely related to the notion of *linearization points* in linearizability proofs, which are points where each method can be seen to atomically satisfy its sequential specification. Linearizability is a strong property that combines atomicity with functional correctness. Therefore, most prior works on linearizability either do not support complete automation [9, 10, 22, 26, 28, 31] or search for linearizability bugs in a bounded number of threads [6,7,24,33,34,36]. Notable exceptions are techniques based on abstract interpretation [2,3,11,32] and observer automata [1]. The first approach [2, 3, 11, 32] instruments each linearization point with the surrounding method's specification and relies on abstract interpretation of the instrumented class to check that the implementation and specification methods always return the same value in the context of the most general client. The second approach [1] instruments each method to generate an *abstract event* whenever a linearization point is passed, captures the specification as an observer automata on the abstract events, and checks the safety of the cross-product of the program and the observer. These approaches are more general than ours and can verify low-level concurrent data structures, but they require explicit reasoning about all possible interactions among the methods of the data structure. Condensability imposes

$$((v, m) = m.get(k)) \Rightarrow (v = m(k) \wedge m' = m)$$

$$(m' = m.put(k, v)') \Rightarrow (m' = m[k \mapsto v])$$

$$\begin{array}{l}(m', v') = m.putIfAbsent(k, v) \Rightarrow \\ \quad v' = m(k) \ \wedge \\ \quad ((m(k) = null) \ \wedge \ (m' = m[k \mapsto v])) \ \vee \\ \quad (\neg(m(k) = null) \ \wedge \ (m = m'))\end{array}$$

Fig. 2. Axioms for `get`, `put` and `putIfAbsent` methods

stronger requirements but in turn enables separate verification of methods of a client class. Finally, a modular set of sufficient conditions for linearizability has been proposed specifically for concurrent queues [17].

2 Example

We now illustrate our approach for automatically verifying atomicity for clients of concurrent data structures through the `AtomicHistogram` example in Figure 1. Our approach verifies the atomicity of each method in the class in isolation; we will illustrate how it works on the `inc` method.

Specifications. We assume that `AtomicMap` is atomic and that we are given specifications for its methods. Figure 2 depicts the axioms characterizing the behavior of the `get`, `put` and `putIfAbsent` methods of a map. The specifications are first-order logic assertions with equality and uninterpreted functions. We model each method as returning a pair of a return value (when the return type is not `void`) and a new map, and we model the abstract map state as a function from each key in the map to its value and from each key not in the map to `null`. We use $m[k \mapsto v]$ to denote the state that maps the key k to the value v and otherwise agrees with the map state m. The axiom for the `putIfAbsent` method states that the mapping of the input key k is updated to the input value v if the previous mapping of k is $null$, and otherwise the map state remains unchanged. The return value of `putIfAbsent` is always the old mapping for the key k.

Purity. To show that the `inc` method in `AtomicHistogram` is atomic, we will show that every possible execution of the method is condensable. Due to the `while` loop there are an unbounded number of execution paths. We address this challenge by leveraging the notion of *purity* from past work on atomicity [13, 35]. At a high level, a loop is pure if only the last iteration of the loop has externally observable effects. If a loop is pure then only the last iteration needs to be considered when reasoning about atomicity, thereby reducing verification of atomicity to loop-free programs. Our approach requires and checks that each loop in a method is pure.

The loop in `inc` in Figure 1 is pure: each loop iteration attempts to write to the map (via either `putIfAbsent` or `replace`) and only continues to iterate if the write fails to happen (`putIfAbsent` returns a non-null value or `replace` returns

```
1 // First path
2 Integer i = m.get(key);
3 assume (i == null);
4 Integer r = m.putIfAbsent(key, 1); ⊛
5 assume (r == null);
6 return 1;
```

```
1 // Second path
2 Integer i = m.get(key);
3 assume (!(i == null));
4 Integer ni = i + 1;
5 Boolean b = m.replace(key, i, ni); ⊛
6 assume (b);
7 return ni;
```

Fig. 3. The two loop-free paths of `inc`

`false`). Given the specifications for the map operations shown above, it is easy to automatically verify the purity of this loop. Since the loop is pure, henceforth we need only consider the two loop-free execution paths shown in Figure 3. We use an `assume` statement to record the choices made at each conditional.

Condensability. Consider the first path shown in Figure 3. Unfortunately, moverness cannot prove the atomicity of the path. Though both of the calls to `get` and `putIfAbsent` indicate that the key is not in the map, other threads can add and then remove the key between `get` and `putIfAbsent`, causing an ABA problem [27]. Using moverness would require that either `get` be a *right mover*, commuting with any subsequent operation from another thread, or that `putIfAbsent` be a *left mover*, commuting with any preceding operation from another thread. However the `get` call does not commute with a subsequent operation from another thread that `puts` the same key into the map. Similarly the `putIfAbsent` call does not commute with a preceding operation from another thread that removes the same key from the map. Although the path is atomic, the moverness requirement is too strong to prove it.

Instead, given an interleaved execution of the client method, condensability identifies a method call on the base atomic object called the *condensation point* and attempts to prove that the interleaved execution of the client method can be replaced by a sequential execution of the client method at the condensation point, which we call the *condensed execution*. We heuristically identify the condensation point as a method call that mutates the state of the underlying concurrent data structure. If the heuristic fails, the static analysis can be repeated for each method call in the path. The heuristically identified condensation points are marked with ⊛ in the paths of Figure 3.

Consider an arbitrary execution X of a concurrent program on a histogram `h` that includes the first path of the `inc` method. We assume the methods of `m` are atomic but make no other atomicity assumptions. Since `m` is atomic, there is some execution S of the program such that S is *equivalent* to X for `m` (i.e. the execution S contains the same method calls and return values on `m` as X) and

S is *sequential* for `m` (i.e. each method call on `m` in S is immediately followed by its associated return). Therefore, the portion of S that includes the execution of the first path has the following form:

```
1 // m0
2 Integer i = m.get(key);
3 // m1
4 // Interleaving (other method calls on m)
5 // m2
6 Integer r = m.putIfAbsent(key, 1); ⊛
7 // m3
```

Here, the states `m0` and `m1` denote the pre-state and post-state of the method call `m.get(key)`, and `m2` and `m3` denote the pre-state and post-state of the method call `m.putIfAbsent(key, 1)` for `m` in S. While S is sequential for the map `m`, it is not necessarily sequential for the histogram `h` due to the interleaving of other method calls from other threads between the calls to `get` and `putIfAbsent`.

To prove the condensability of this execution of `inc`, we must prove the following conditions:

1. None of the method calls other than the condensation method call mutate the state of `m`.
2. Consider a *condensed* execution of `inc` from the condensation point, that is, a sequential execution of `inc` starting from the state `m2` for the map `m`.

```
1 // m2
2 Integer result = h.inc(key);
3 // m3'
```

   2.1. The state of the map after the condensed execution should be the same as the post-state of the condensation method call.
   2.2. The two calls to `inc` should have the same return value.

The first condition above requires us to prove that `m0` = `m1`, which is easily discharged given our earlier specification for `get`. The second condition requires us to reason about the execution path taken by the condensed execution of `inc` which in general can differ from the path taken in the original execution. Since in the original execution, the call to `putIfAbsent` from state `m2` returns `null`, it is easily seen using the specifications for `get` and `putIfAbsent` that the condensed execution of `inc` will look as follows:

```
1 // m2
2 Integer i' = m.get(key);
3 // m2
4 Integer r' = m.putIfAbsent(key, 1);
5 // m3'
6 return 1;
```

Specifically, the call to `get` will return `null`, so the "then" branch at line 11 in `inc` will be executed. Therefore `putIfAbsent` is called from the same state `m2` as in the original execution, so the (assumed) determinism of `putIfAbsent` implies

that `m3` = `m3'`, discharging condition 2.1. Finally, condition 2.2 is trivial in this case, since both executions of `inc` end with the statement `return 1`.

A similar analysis can be done to show that the second path in Figure 3 is also condensable, and hence that `inc` is condensable. Note that this analysis is completely *modular*: the condensability of `inc` can be proven without having to explicitly enumerate the possible interactions with the other methods in the histogram class, or even to know the full set of such methods.

If each method in the histogram is condensable, then we say that the histogram itself is condensable. In the next section we formalize the notion of condensability and show that condensability implies atomicity.

3 Atomicity and Condensability

In this section, we first present some preliminary definitions and formalize the standard notion of atomicity. Then we define condensability and state our main theorem, that condensability implies atomicity.

3.1 Executions and Atomicity

Method Calls and Events. Let l, o, n, T, and v denote a label, an object, a method name, a thread and a value. Let $inv(l \triangleright o.n_T(v))$ denote an invocation event of a method call labeled l by thread T that calls the method n on the object o with the argument v. Let $ret(l \triangleright v)$ denote a response event of the method call labeled l that returns v.

Operations on event sequences. Let E and E' be event sequences. For a thread T, we use $E|T$ to denote the subsequence of all events of T in E. For an object o, we use $E|o$ to denote the subsequence of all events of o in E.

Executions. An *execution* X is a sequence of events where each invocation event has a unique label and every thread T is well-formed in X (i.e. $X|T$ is an alternating sequence of invocations and responses, with each pair of an invocation and response having the same label). We say label l is in X and write $l \in X$ if there is an invocation event with label l in X. Let $Labels(X)$ denote the set of labels in X. The functions iEv and rEv on $Labels(X)$ map a label to the invocation and the response events associated with the label.

An execution X is *equivalent* to an execution X' if one is a permutation of the other one; that is, only the events are reordered but the components of the events (including the argument and return values) are preserved.

Real-time relations. For an execution X, we define the real-time relations $\prec_X$, and $\preceq_X$ on $Labels(X)$ as follows: $l_1 \prec_X l_2$ if and only if $rEv(l_1)$ precedes $iEv(l_2)$ in X, and $l_1 \preceq_X l_2$ if and only if $l_1 \prec_X l_2 \ \vee \ l_1 = l_2$.

An execution X is sequential iff $\forall l, l' \in X\colon l \preceq_X l' \ \vee \ l' \preceq_X l$.

Definition 1 (Atomicity). *An execution X of a program p is* atomic *for an object o if and only if there exists an execution S of p (called the* justifying execution of X for o*) such that*

– $S|o$ *is sequential,*
– $S|o$ *is equivalent to* $X|o$*, and*
– $S|o$ *is real-time-preserving i.e.* $\prec_{X|o} \subseteq \prec_{S|o}$.[1]

An object o is atomic iff every execution of every program is atomic for o.

Atomicity considers sequential executions on the object as justifying executions. On the other hand, linearizability requires the justifying execution to be a member of a pre-defined sequential specification for the object. In other words, an atomic object is linearizable with respect to its sequential executions.

3.2 Condensability

Now we can define condensable objects and state our condensability theorem.

A method call on an object o is an *accessor* if it does not change the state of o, and otherwise the method is a *mutator*. For example, a call to `putIfAbsent` is a mutator if it returns `null` and is an accessor otherwise. We say that an object c *composes* object o if the only shared object in the implementation of c is o; any other object accessed by methods of c is either local or thread-local.

The following definition formalizes the notion of condensability that we informally described in the previous section.

Definition 2 (Condensable). *Consider an object c that composes an atomic object o. A method m of c is* condensable *if and only if for every execution X and justifying execution S of X for o, and for every execution e of m in S, there exists a method call $\mathcal{P}(e)$ on o in e such that*

1. *All the method calls on o in e other than $\mathcal{P}(e)$ are accessors.*
2. *Let s be the sequential execution of m with the same arguments as in e and the same pre-state for o as $\mathcal{P}(e)$ in S,*
   2.1. *s results in the same post-state for o as $\mathcal{P}(e)$ in S, and*
   2.2. *s results in the same return value as e.*

The method call $\mathcal{P}(e)$ is called the condensation point *and the execution s is called the* condensed execution*. An object is condensable if and only if all of its methods are condensable.*

Note that the condensed execution s of m may take a different path from the original execution e.

A notable property of the above definition is that the condensability of a method is independent of that of other methods. This independence supports modular verification of condensability for each method of an object.

The following theorem states our main result.

Theorem 1 (Condensability). *Every condensable object is atomic.*

[1] Real-time-preservation is often implicitly assumed.

Please see the technical report [21] for the proof. Let us intuitively explain why the condensability conditions are sufficient for atomicity. Consider an arbitrary execution X of a program on c. As o is atomic, there is a justifying execution S of X for the atomicity of o. Our goal is to produce a justifying execution S' for the atomicity of c. The idea is to construct the execution S' from S as follows: every execution of a method call on c is removed from S and replaced by its condensed execution at its condensation point.

By construction, no two method calls on c interleave in S'; thus, $S'|c$ is sequential. To prove that S' is real-time-preserving for c, we need to show that if a method call m_1 on c with execution e_1 is before a method call m_2 on c with execution e_2 in X, then m_1 is before m_2 in S'. As e_1 is before e_2 in X, $\mathcal{P}(e_1)$ is before $\mathcal{P}(e_2)$ in X. We have that S is real-time-preserving for o thus, as $\mathcal{P}(e_1)$ is before $\mathcal{P}(e_2)$ in X, we have that $\mathcal{P}(e_1)$ is before $\mathcal{P}(e_2)$ in S as well. Thus, by the construction of S', m_1 is before m_2 in S'.

Therefore, it remains to show that S' is an execution of the program that is equivalent to X. Consider two consecutive condensed executions s_1 and s_2 in S' that replace two condensation methods calls m_1 and m_2 in S. To prove that S' is an execution of the program, we should show that the state of o in the post-state of s_1 is equal to the state of o that is assumed in the pre-state of s_2. This fact is derived from the following three equalities. First, by condition 2.1 above the state of o in the post-state of s_1 is equal to the state of o in the post-state of m_1. Second, since there is no condensation method call between m_1 and m_2 and by condition 1 all the other method calls on o are accessors, the state of o in the post-state of m_1 is equal to the the state of o in the pre-state of m_2. Third, by construction the state of o in the pre-state of s_2 is equal to the state of o in the pre-state of m_2. Finally, to complete the proof that S' is equivalent to S we leverage condition 2.2 above, which requires each call in S' to have the same return value as its counterpart in X.

4 Checking Condensability

In this section, we show how condensability of a loop-free client method can be represented as constraints and automatically checked. We assume that all method calls in the client method are on the underlying atomic data structure. We will relax this assumption in the next section.

Consider a loop-free client method with the input parameter p. Let o be the underlying atomic data structure. Let $\overline{P}$ be the set of paths of the method. Let P_i denote the ith path. Let $|P|$ denote the size of P. Let us denote a path with the triple $(b, \overline{m}, r)$ where b is the conjunction of the branch conditions of the path, $\overline{m}$ is the sequence of method calls $y = o.n(x)$ of the path and r is the returned variable of the path. In the sequence of method calls $\overline{m}$, let m_k denote the kth method call. Let $|\overline{m}|$ denote the size of $\overline{m}$. See the technical report [21] for how we compute the paths.

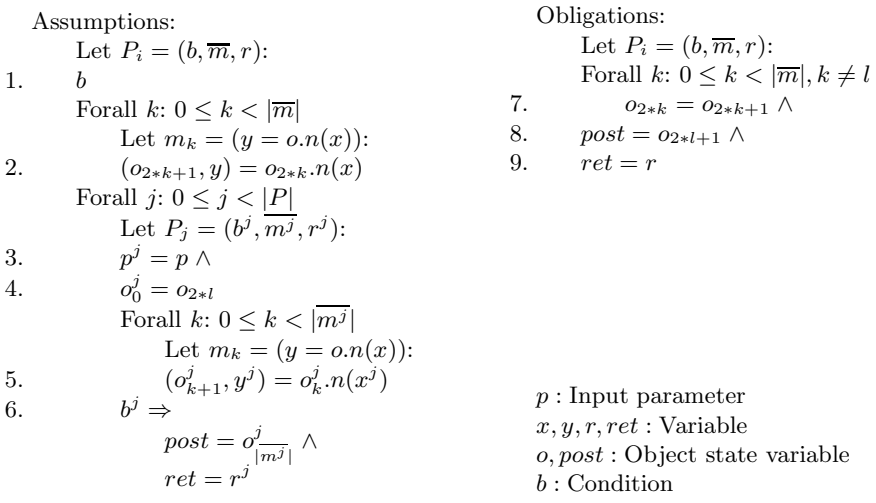

Fig. 4. Checking Condensability of the ith path at its lth method call

We check the condensability of each path separately. Let us focus on the ith path $P_i = (b, \overline{m}, r)$. The condensation point of a path is one of its method calls. Let us consider the condensability of the ith path at its lth method call. We want to generate assumptions and obligations that verify that for every execution X and justifying execution S of X for o, for every execution of the ith path in S, the method call m_l is the condensation point. We consider an arbitrary execution X and an arbitrary justifying execution S of X for o. We assume that the ith path is executed in S. The set of assumptions and obligations to check the condensability of the ith path at the lth method call is depicted in Figure 4. We describe each of them in turn.

To indicate that the ith path is executed in S, we assert the branch conditions of the path (line 1) and assert that each method call on the path is executed (line 2). The assertion $(o_2, y) = o_1.n(x)$ denotes a method call n on o with pre-state o_1 and argument x that results in post-state o_2 and return value y. The pre-state and post-state variables of the kth method are o_{2*k} and o_{2*k+1} respectively. Note that due to arbitrary interleaving with other threads, the method calls of the path are not necessarily adjacent in S. Therefore, the post-state variable o_{2*k+1} of the kth method call is different from the pre-state variable $o_{2*(k+1)}$ of the $(k+1)$th method call in the path.

Next, we represent the condensed execution s of the client method at the condensation point. The condensed execution could take any of the possible paths through the client method, so we must consider all of them. The states and variables of each path are superscripted with the index of the path, so that they do not conflict with one another. Consider one such path P_j. First we assert that the input parameter to the condensed execution is equal to the input value of the original method execution (line 3). Next we assert that the pre-state of

the condensed execution o_0^j is equal to the pre-state of the condensation point o_{2*l} (recall that the condensation point is the lth method call in the original path) (line 4). Finally the method calls of the condensed execution are asserted (line 5). Note that since the condensed execution s is sequential, the post-state of each method call is the same as the pre-state of the subsequent method call.

Next, we identify which path is actually taken by the condensed execution. Specifically, the path taken is the unique path whose branch conditions are satisfied. Therefore, line 6 has the effect of equating *post* to the post-state of the condensed execution and *ret* to the return value of the condensed execution.

Finally, we present the proof obligations for condensability of the ith path. All the method calls in the ith path other than the condensation point must be accessors i.e., their pre and post-states must be equal (line 7). The post-state of the condensed execution path *post* must be equal to the post-state of the condensation method call o_{2*l+1} in the ith path (line 8). The return value of the condensed execution *ret* must equal the return value of the ith path (line 9).

As an example, we present the constraints that each line of Figure 4 generates for the first path of the `inc` method in Figure 3. Line 1 generates $i = null \ \wedge \ r = null$. Line 2 generates $(m_1, i) = get(m_0, key)$ and $(m_3, r) = putIfAbsent(m_2, key, 1)$. For the first path of the `inc` method, line 3 generates $key^0 = key$, line 4 generates $m_0^0 = m_2$, line 5 generates $(m_1^0, i^0) = get(m_0^0, key^0)$ and $(m_2^0, r^0) = putIfAbsent(m_1^0, key^0, 1)$, and line 6 generates $(i^0 = null \wedge r^0 = null) \Rightarrow (post = m_2^0 \ \wedge \ ret = r^0)$. Similar constraints are generated for the second path. The proof obligations are as follows: Line 7 generates $m_0 = m_1$. Lines 8 and 9 generate $post = m_3$ and $ret = r$.

Note that in Figure 4, the universal quantification can be expanded. Therefore, the assumptions and proof obligations are quantifier-free formulas that an SMT solver can discharge automatically.

5 Snowflake

Now we present our tool called Snowflake that automatically verifies condensability of Java methods.

User Input. The user must provide Snowflake with the client method to check along with the axioms that characterize the methods of the data structure used by the client method. The user also specifies the variable/field in the client code that holds the underlying data structure object with the `BaseObject` Java annotation. Finally, Snowflake supports optional annotations to declare that a certain method call in the client method is *functional*, meaning that the call is side-effect-free and that its return value is solely a function of the states of the given receiver object and arguments. A variation on this annotation declares a method call to be *argument-functional*, which is identical except that the method's return value does not depend on the receiver object's state. These annotations allow Snowflake to verify condensability modularly, without having to recursively analyze calls to auxiliary methods in the given client method.

We presented the axioms for the `get`, `put`, and `putIfAbsent` methods of the atomic map in Figure 2. The documentation of current data structures typically presents a pseudocode specification for the *conditional* atomic methods in terms of the more basic methods. For example, the sequential specification of `putIfAbsent` in terms of `get` and `put` methods is depicted in Figure 5. The axiom of `putIfAbsent` in Figure 2 can be derived from its sequential specifications in Figure 5 along with the axioms of the `get` and `put` methods in Figure 2. Our tool has embedded axioms for common methods of Java concurrent map and set data structures and can be extended to support other collection types. We present the full set of axioms in the technical report [21].

```
1 V putIfAbsent(K k, V v) {
2   atomic {
3     V v1 = get(k);
4     if (v1 == null)
5       put(k, v);
6     return v1;
7   }
8 }
```

Fig. 5. The specification of `putIfAbsent` in terms of `get` and `put`

Paths and Purity. As the first step, Snowflake computes the set of paths of the client method. We adopt the terminology of paths from [13] and [35]. A path of a loop is *exceptional* if it is executed as the last iteration of the loop. An exceptional loop path ends in a break or return statement or by the condition of the loop evaluating to false. A path of a loop is *normal* if it is not exceptional. Informally, a loop is pure if its normal paths have no side effects. We conservatively determine a loop to be pure if for every method call $y = o.n(x)$ in a normal path of the loop:

- If o is a shared variable, then the method call is an accessor.
- The variable y is a local variable.
- For all paths in the control flow graph from the end of this normal path to the return of the method call, the next access to y, if any, overwrites it.

For example, the method calls in the normal paths of the `inc` method of Figure 1 satisfy these conditions.

An *exceptional variant* of a method is a copy of the method where each pure loop of the method is replaced by one of its exceptional paths. The exceptional variant of an object is the copy of the object where each method is replaced by all of its exceptional variants. The following theorem is a restatement of Theorem 5.2 from [35]:

Theorem 2. *If the exceptional variant of an object is atomic, then the object is atomic.*

The theorem reduces verification of atomicity for methods with pure loops to loop-free methods.

Given a client method, Snowflake computes the normal paths of the loops and the exceptional variants of the method. It then converts each path to its static single assignment (SSA) form. It first checks the purity of the normal paths using the conditions described above. If the purity of a normal path cannot be verified, the client method is rejected.

We check the condensability of each exceptional variant using the method described in Section 4. We use the following heuristics to guess the condensation point of a path. If there is a call to a method that can mutate the data structure's state in the path, the condensation point is the last such method; otherwise, it is the first method on the data structure in the path. As mentioned before, if the heuristic fails, we can iterate our approach with a different method call as the condensation point.

Now, let us relax the assumption that all the method calls in a path are on the atomic data structure. If there is a method call that is not on the atomic data structure and is not annotated as functional, the client method is rejected because we cannot modularly ensure atomicity. Otherwise, we treat each functional method as an uninterpreted function. Specifically, a functional method call $y = o.n(x)$ is translated to the assertion $y = n(o, x)$. Therefore, as long as the ith path and the condensed execution call such a method with equal arguments, we can prove that they will have equal results. Mathematical operations are treated similarly but we additionally assert axioms such as commutativity and associativity.

Snowflake represents all of the assertions and obligations, along with axioms for the atomic data structure in the SMT2 format and invokes the Z3 SMT solver to check their validity. A method is considered condensable if this process succeeds for each of the method's exceptional variants. An object is considered condensable if each of its methods is found to be condensable.

6 Results

Benchmarks and Platform. We adopt the benchmark suite available from Colt [30]. This benchmark suite is a collection 112 client methods from 51 real-world applications such as Apache Tomcat, Cassandra, and MyFaces Trinidad. We call this collection the Colt suite. It consists of 26 atomic and 86 non-atomic methods.

Snowflake is written in Java, compiled and executed with JDK version 1.7.0.07 and uses Polyglot [25] version 2.5.1 and Z3 version 4.3.2 [8]. The source code of Snowflake is available [21].

Results. Snowflake is sound, so it correctly rejects all 86 non-atomic benchmarks in the Colt suite. Figure 6 shows the result of applying Snowflake to the 26 atomic benchmarks of Colt suite. The pie chart partitions these benchmarks into three groups. Twenty of the benchmarks (76.9%) are proven atomic without any change. The other six benchmarks are rejected as non-atomic by Snowflake.

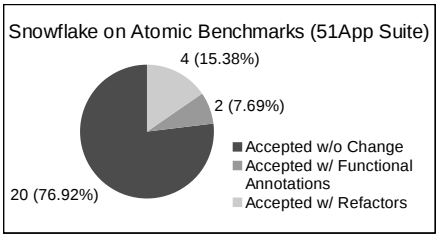

Fig. 6. Evaluation of Snowflake

However, with small modifications they can also be proven to be atomic: two of them simply require the addition of functional annotations on some methods, and the other four benchmarks require some code refactoring. For example, we refactored a block of code that initializes a new object to a method call and annotated the method call as functional. Snowflake verified each of the 26 atomic benchmarks in an average time of 1.45 seconds with a minimum of 1.16 seconds and a maximum of 2.66 seconds. We present a list of benchmarks and the run times of our tool in the accompanying web page [21].

A comparison with Colt is instructive. Since Snowflake is a verification tool, if it accepts a method, it is atomic, but if it rejects the method, it may still be atomic. On the other hand, since Colt is a bug-finding tool, if it rejects a method, it is non-atomic, so it does not find any atomicity errors in the 26 atomic benchmarks of Colt suite. However Colt may not reject non-atomic benchmarks and indeed Colt is not able to find atomicity errors in four of the non-atomic benchmarks. Each of these benchmarks first atomically gets the current value or puts a value for a key and then performs a separate operation on the value. Although each operation is atomic, they are not atomic together.

7 Conclusion

We introduced condensability as a modular verification technique for atomicity of clients of concurrent data structures. We defined the notion of condensability and proved that it implies atomicity. Condensability of an object can be separately checked for each method of the object. We showed how condensability of a method can be represented as constraints and automatically checked. We presented our tool, Snowflake, that automatically verifies condensability and applied it to real-world client methods. In future work, we are interested to generalize our approach to support impure loops as well as multiple writes to the data structure.

Acknowledgements. Thanks to Madan Musuvathi and Erez Petrank for initial discussions on this topic, and to Lorenzo Gomez for contributions to the Snowflake tool.

References

1. Abdulla, P.A., Haziza, F., Holík, L., Jonsson, B., Rezine, A.: An integrated specification and verification technique for highly concurrent data structures. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 324–338. Springer, Heidelberg (2013)
2. Amit, D., Rinetzky, N., Reps, T., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 477–490. Springer, Heidelberg (2007)
3. Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M.: Thread quantification for concurrent shape analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 399–413. Springer, Heidelberg (2008)
4. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency control and recovery in database systems, vol. 370. Addison-wesley, New York (1987)
5. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: Transactional predication: High-performance concurrent sets and maps for stm. In: Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC 2010, pp. 6–15. ACM, New York (2010)
6. Burckhardt, S., Dern, C., Musuvathi, M., Tan, R.: Line-up: A complete and automatic linearizability checker. In: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, pp. 330–340. ACM, New York (2010)
7. Burckhardt, S., Gotsman, A., Musuvathi, M., Yang, H.: Concurrent library correctness on the tso memory model. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 87–107. Springer, Heidelberg (2012)
8. de Moura, L., Bjørner, N.S.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
9. Derrick, J., Schellhorn, G., Wehrheim, H.: Verifying linearisability with potential linearisation points. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 323–337. Springer, Heidelberg (2011)
10. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 97–114. Springer, Heidelberg (2004)
11. Drăgoi, C., Gupta, A., Henzinger, T.A.: Automatic linearizability proofs of concurrent objects with cooperating updates. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 174–190. Springer, Heidelberg (2013)
12. Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, pp. 2–15. ACM, New York (2009)
13. Flanagan, C., Freund, S.N., Qadeer, S.: Exploiting purity for atomicity. IEEE Trans. Software Eng. 31(4), 275–291 (2005)
14. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI 2003, pp. 338–349. ACM, New York (2003)
15. Flanagan, C., Qadeer, S.: Types for atomicity. In: Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, TLDI 2003, pp. 1–12. ACM, New York (2003)
16. Golan-Gueta, G., Ramalingam, G., Sagiv, M., Yahav, E.: Concurrent libraries with foresight. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming

Language Design and Implementation, PLDI 2013, pp. 263–274. ACM, New York (2013)
17. Henzinger, T.A., Sezgin, A., Vafeiadis, V.: Aspect-oriented linearizability proofs. In: D'Argenio, P.R., Melgratti, H. (eds.) CONCUR 2013 – Concurrency Theory. LNCS, vol. 8052, pp. 242–256. Springer, Heidelberg (2013)
18. Herlihy, M., Koskinen, E.: Transactional boosting: A methodology for highly-concurrent transactional objects. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2008, pp. 207–216. ACM, New York (2008)
19. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
20. Jonsson, B.: Using refinement calculus techniques to prove linearizability. Formal Aspects of Computing 24(4-6), 537–554 (2012)
21. Lesani, M., Millstein, T., Palsberg, J.: Automatic atomicity verification for clients of concurrent data structures, the companion, `http://www.cs.ucla.edu/~lesani/companion/cav14`, `http://fmdb.cs.ucla.edu/tech_reports/default.lasso`
22. Liang, H., Feng, X.: Modular verification of linearizability with non-fixed linearization points. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, pp. 459–470. ACM, New York (2013)
23. Lipton, R.J.: Reduction: A method of proving properties of parallel programs. Commun. ACM 18(12), 717–721 (1975)
24. Liu, Y., Chen, W., Liu, Y.A., Sun, J.: Model checking linearizability via refinement. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 321–337. Springer, Heidelberg (2009)
25. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An extensible compiler framework for java. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 138–152. Springer, Heidelberg (2003)
26. O'Hearn, P.W., Rinetzky, N., Vechev, M.T., Yahav, E., Yorsh, G.: Verifying linearizability with hindsight. In: Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC 2010, pp. 85–94. ACM, New York (2010)
27. International Business Machines Corporation. Product Publications. IBM System/370 Extended Architecture: Principles of Operation. IBM Corporation (1983)
28. Schellhorn, G., Wehrheim, H., Derrick, J.: How to prove algorithms linearisable. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 243–259. Springer, Heidelberg (2012)
29. Shacham, O.: Verifying Atomicity of Composed Concurrent Operations. PhD thesis, Tel Aviv University (2012)
30. Shacham, O., Bronson, N., Aiken, A., Sagiv, M., Vechev, M., Yahav, E.: Testing atomicity of composed concurrent operations. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2011, pp. 51–64. ACM, New York (2011)
31. Vafeiadis, V.: Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory (July 2008)
32. Vafeiadis, V.: Automatically proving linearizability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 450–464. Springer, Heidelberg (2010)

33. Černý, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., Alur, R.: Model checking of linearizability of concurrent list implementations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 465–479. Springer, Heidelberg (2010)
34. Vechev, M., Yahav, E.: Deriving linearizable fine-grained concurrent objects. In: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2008, pp. 125–135. ACM, New York (2008)
35. Wang, L., Stoller, S.D.: Static analysis of atomicity for programs with non-blocking synchronization. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2005, pp. 61–71. ACM, New York (2005)
36. Zhang, S.J.: Scalable automatic linearizability checking. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, pp. 1185–1187. ACM, New York (2011)
37. Zomer, O., Golan-Gueta, G., Ramalingam, G., Sagiv, M.: Checking linearizability of encapsulated extended operations. In: Shao, Z. (ed.) ESOP 2014 (ETAPS). LNCS, vol. 8410, pp. 311–330. Springer, Heidelberg (2014)