Line-Up: A Complete and Automatic Linearizability Checker

Sebastian Burckhardt
Microsoft Research
sburckha@microsoft.com

Chris Dern
Microsoft
chrisd@microsoft.com

Madanlal Musuvathi
Microsoft Research
madanm@microsoft.com

Roy Tan
Microsoft
roytan@microsoft.com

Abstract

Modular development of concurrent applications requires thread-safe components that behave correctly when called concurrently by multiple client threads. This paper focuses on linearizability, a specific formalization of thread safety, where all operations of a concurrent component appear to take effect instantaneously at some point between their call and return. The key insight of this paper is that if a component is intended to be deterministic, then it is possible to build an automatic linearizability checker by systematically enumerating the sequential behaviors of the component and then checking if each its concurrent behavior is equivalent to some sequential behavior.

We develop this insight into a tool called Line-Up, the first complete and automatic checker for *deterministic linearizability*. It is complete, because any reported violation proves that the implementation is not linearizable with respect to *any* sequential deterministic specification. It is automatic, requiring no manual abstraction, no manual specification of semantics or commit points, no manually written test suites, no access to source code.

We evaluate Line-Up by analyzing 13 classes with a total of 90 methods in two versions of the .NET Framework 4.0. The violations of deterministic linearizability reported by Line-Up exposed seven errors in the implementation that were fixed by the development team.

Categories and Subject Descriptors D [2]: 4

General Terms Algorithms, Reliability, Verification

Keywords Linearizability, Atomicity, Thread Safety

1. Introduction

Concurrent programming is becoming more prevalent as Moore's law pays fewer dividends for sequential applications. To achieve modular development, programmers face the question of how to build and test *thread-safe* components, that is, components that function correctly in a concurrent environment without placing undue synchronization burden on the caller.

The most well-known examples of thread-safe components are concurrent data types such as queues or sets, which are provided by many major concurrency libraries (java.util.concurrent, Intel Threading Building Blocks, and Microsoft .NET 4.0). Such libraries can simplify the development of thread-safe code, but are themselves difficult to develop and test. Also, because such libraries cannot cover all scenarios, we expect that a growing number of programmers will develop concurrent components that are tailored to their applications.

Thread-safety is a vaguely defined term; more specific correctness conditions include data-race-freedom [11, 17, 24], serializability [23] (sometimes called atomicity [10, 12]), linearizability [15], or sequential consistency [16]. Over the past two decades, research on concurrent data types confirmed that fine-grained concurrency is difficult to get right (by discovering bugs in published algorithms [6, 18]) and researchers have focused on linearizability as a standard for correctness [14, 26].

A concurrent component is linearizable if its operations, when called concurrently, appear to take effect instantaneously at some point between their call and return. As a result, one can understand the concurrent behavior of a linearizable component simply by understanding its sequential behavior. One way to design a linearizable component is to protect all instructions in an operation by a single lock in the component. For performance reasons, many concurrent components, in practice, use more sophisticated lock-free synchronization to guarantee linearizability.

For such implementations, correctness is subtle enough to warrant manual proofs of linearizability. Unfortunately, such proofs are labor-intensive, difficult to apply to detailed production code, and require formal training. On the other hand, promising experiences with model checking [2, 27] suggest that a focus on falsification and precise counterexamples is a viable alternative, and may reach a wider audience by virtue of better automation and a more productive user experience.

The key challenge for an automatic linearizability checker is that the linearizability of a concurrent component is defined with respect to a sequential specification. Asking the user to provide such a specification would depart unacceptably from our goal of automation. Not only are most users unfamiliar with formal specifications, but the act of writing a specification is labor-intensive even for experts.

Our insight is that if the sequential specification is deterministic, it is possible to automatically generate the specification by systematically enumerating all sequential behaviors of the component. We use this insight to build a tool called Line-Up that can find violations of deterministic linearizability *automatically*, without requiring knowledge of the implementation, source annotations, or even source code.

Line-Up uses a stateless model checker that runs the same implementation both sequentially and concurrently, and checks the consistency of the observations. Line-Up is complete: Any detected violation of deterministic linearizability is a conclusive proof that the implementation is not linearizable with respect to *any* deterministic specification. On the other hand, Line-Up (like all dynamic checking tools) is sound only with respect to the inputs and the executions tested.

As a second contribution, we tightened the original definition of linearizability to additionally detect erroneous blocking of implementations. This improves the coverage of our method as it allows us to find liveness errors in implementations. We also describe an adaptation of our algorithm that uses random sampling to find bugs more quickly.

We evaluate Line-Up on production code, specifically the concurrent data types that ship with Microsoft's .NET Framework 4.0, and that were made available in two prereleases (community technology preview, and beta release). In the course of our project, we applied Line-Up to 13 classes with a total of 90 methods. We found 12 root causes for violations of deterministic linearizability. Of those, 7 were caused by real implementation errors. The other 5 revealed behavior that was intentionally non-linearizable or nondeterministic. Considering that we tested 90 methods, nondeterminism was rather rare (somewhat contrary to the prominent use of nondeterministic specifications in the research literature). In some cases the developers realized that a method is nondeterministic only after the fact was detected by Line-Up, and updated the documentation.

To test our choice of linearizability as the appropriate notion for thread safety, we evaluated Line-Up with other correctness checkers such as data-race detection and conflict serializability. Our experiments revealed that these were not well suited for this application: data-race detection was ineffective because the code contained only benign data races (due to a disciplined use of 'volatile' qualifiers and interlocked operations), while conflict-serializability checking produced a discouraging number of false alarms. The implementations we studied contained a large variety of programming patterns that violate conflict serializability, but are nevertheless correct.

1.1 Example

To illustrate how Line-Up operates, consider a situation where we would like to test a concurrent queue using a black-box approach (say, we do not have access to the source code). First, we specify a set of method calls that we would like to test. For instance, after inspecting the interface of the queue, we might consider testing with two methods that add different values to the queue and one method that removes a value:

$$\{\mathit{queue.Add}(200), \quad \mathit{queue.Add}(400), \quad \mathit{queue.TryTake}()\}.$$

This is the only manual step required when using Line-Up. Line-Up then automatically enumerates concurrent and sequential combinations of these operations. If it finds a violation of deterministic linearizability, it reports a small concurrent test scenario such as shown in Fig. 1. In fact, this example was a violation of deterministic linearizability that exposed a real bug in the .NET 4.0 community technology preview [19]. A client of the queue implementation would expect a *TryTake* to fail only when the queue is empty. However, the buggy behavior shown in Figure 1 was caused by accidentally allowing a lock acquire in *TryTake* to time out.

Note that the problem in this example can be intuitively understood without referring to implementation details, and without knowing the formal definition of linearizability. This is an important advantage when trying to convince developers that their implementation has a bug.

Thread 1	Thread 2
queue.Add(200);	*queue.Add*(400);
queue.TryTake() → returns 200;	*queue.TryTake*(); → FAILS

Figure 1. A buggy queue implementation that violates linearizability. In the intended behavior of a queue, both TryTake operations should succeed, even though they can return different values depending on the interleaving of these operations. Line-Up is able to automatically detect this violation in the example above.

1.2 Related Work

To achieve full verification of concurrent objects, researchers construct linearizability proofs, using rely-guarantee reasoning [25, 26] or simulation relations (using both forward- and backward-simulations) [4]. These proof methods are interactive, not automatic, and require sequential specifications. In addition, many of these techniques require the user to specify *linearization points*, which are points in the implementation where the operations appear to instantaneously take effect. The location of these linearization point may depend on conditions that are not statically known, requiring nontrivial annotations [27]. Even worse, those conditions may depend on future events, thus requiring the use of prophecy variables [1] or backward simulation relations [4]. Line-Up avoids these annotations and instead considers all possible linearization points, following the original definition of linearizability [15].

Model checking is more automatic and has the advantage of producing counterexamples, but usually verifies only bounded executions or finite-state models. We know of two efforts in this area, but neither achieves the same automation as Line-Up or applies the method to real production code: (1) CheckFence [2] uses a two-phase check similar to ours, but does not check linearizability and requires the user to write a test suite manually. (2) An experience report with the model checker SPIN [27] briefly mentions an automatic procedure, but focuses mainly on manually annotated linearization points; it also appears to operate on finite-state abstract models rather than on full-featured code.

Prior work on runtime refinement checking [8, 9] is closely related to linearizability checking, and does scale to production code. However, it is less automatic than Line-Up as it requires the user to annotate linearization points and to construct the test scenarios.

Another similar approach [13] executes random tests to produce logs, then checks those logs for linearizability using a greedy search (thus avoiding linearization points annotations). The tests are not driven systematically by a model checker, however, and the abstract state is modelled manually.

1.3 Contributions

In the remainder of this paper, we elaborate on our contributions in the following order:

1. We present both the original definition of linearizability as well as a tightened version that can detect erroneous blocking of implementations (Section 2).
2. We present an algorithm that can automatically check whether an implementation is linearizable with respect to *any* deterministic sequential specification (Section 3).
3. We describe how we implemented our algorithm on top of a stateless model checker, and how we incorporated random sampling (Section 4).

4. We demonstrate how our tool found seven real bugs in production code, and compare its effectiveness with dynamic race detection and atomicity checking (Section 5).

Our results show that automatic checking of deterministic linearizability with Line-Up provides an effective way of catching concurrency bugs without producing large numbers of false alarms.

2. Formulation

In this section, we begin with a formal definition of linearizability, closely following the original [15] with the exception of a few minor adaptations[1] (Section 2.1).

We then elaborate on how to compare implementations to specifications using classic linearizability, and why the latter can not detect erroneous blocking of the implementation (Section 2.2). Next, we introduce a generalized definition of linearizability that can check the blocking behaviors of the implementation against the blocking behaviors of the specification (Section 2.3). Finally, we formally define *deterministic linearizability*, the property that Line-Up is checking (Section 2.4).

2.1 Linearizability

For the purposes of this paper, the system is modelled as a set $T = \{A, B, C, \ldots\}$ of sequential threads and a set $O = \{o, p, q \ldots\}$ of objects. Formally, these sets are static, but dynamic thread and object creation can easily be modelled. Each object has a type that defines a set of primitive operations, and these operations are the only means of reading or modifying the state of the object. We represent the operations on an object $o \in O$ by two disjoint sets I_o and R_o, where I_o is the set of invocations (which include the operation name and arguments) and R_o is the set of responses (which may include return values).

For example, consider a concurrent counter object c supporting operations to increment, decrement, set, or get the current count. In this case, we define the invocation and response sets as follows:

$$\begin{aligned} I_c &= \{\mathit{inc}, \mathit{dec}, \mathit{get}\} \cup \{\mathit{set}(x) \mid x \in \mathbb{N}\} \\ R_c &= \{\mathit{ok}\} \cup \{\mathit{ok}(x) \mid x \in \mathbb{N}\} \end{aligned}$$

2.1.1 Histories

An execution of the system is represented as a *history*, which is defined to be a finite sequence of events, where an event is defined as a tuple $\langle o\ a\ t\rangle$ where $o \in O$ is an object, $a \in I_o \cup R_o$ is an invocation or response, and $t \in T$ is a thread. A *subhistory* of a history H is a subsequence of the events of H. We call an event $\langle o\ a\ t\rangle$ a *call* or a *return* depending on whether $a \in I_o$ or $a \in R_o$. A return $\langle o\ a\ t\rangle$ is said to *match* a call $\langle o'\ a'\ t'\rangle$ if $o = o'$ and $t = t'$. Within a history H, a call is pending if it is not followed by a matching return. A history H is *complete* if it contains no pending calls. For a history H, we define $complete(H)$ to be the history obtained from H by deleting all pending calls. A *single-object* history is a history all of whose events are associated with the same object.

For a thread t, we define the *thread subhistory* $H|t$ to be the subhistory of H consisting of all events associated with t. A history H is *serial* if (1) the sequence (if not empty) starts with a call event, (2) calls and returns alternate in the sequence, and (3) each return matches the immediately preceding call. A history H is *well-formed* if the thread subhistory $H|t$ is serial for each thread t. All histories considered in this paper are assumed to be well-formed. See Fig. 2 for an example.

[1] We renamed "processes" as "threads", renamed "sequential" histories to "serial" histories, renamed "invocation" and "response" events to "call" and "return" events, and decomposed the linearizability definition using the notion of a serial witness.

H	$H\|A$	$H\|B$
$\langle c\ \mathit{set}(0)\ A\rangle$	$\langle c\ \mathit{set}(0)\ A\rangle$	$\langle c\ \mathit{get}\ B\rangle$
$\langle c\ \mathit{get}\ B\rangle$	$\langle c\ \mathit{ok}\ A\rangle$	$\langle c\ \mathit{ok}(0)\ B\rangle$
$\langle c\ \mathit{ok}\ A\rangle$	$\langle c\ \mathit{inc}\ A\rangle$	$\langle c\ \mathit{get}\ B\rangle$
$\langle c\ \mathit{inc}\ A\rangle$		$\langle c\ \mathit{ok}(1)\ B\rangle$
$\langle c\ \mathit{ok}(0)\ B\rangle$		
$\langle c\ \mathit{get}\ B\rangle$		
$\langle c\ \mathit{ok}(1)\ B\rangle$		

Figure 2. An example of a well-formed single-object history H (on the left) and two of its thread subhistories (on the right.)

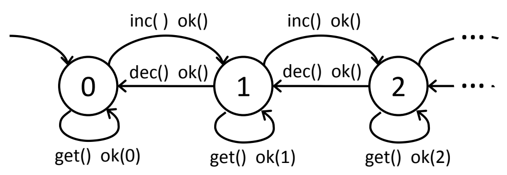

Figure 3. Specification automaton for the counter object.

2.1.2 Sequential Specifications

A set Y of histories is prefix-closed if, whenever H is in Y, every prefix of H is also in Y. A *sequential specification* for an object is a prefix-closed set of single-object serial histories for that object. It can be convenient and illustrative to think of Y as the set of traces generated by some suitably defined specification automaton, such as shown in Fig. 3.

A sequential specification defines the intended semantics because it specifies (1) what values may returned by each operation, and (2) what operations may proceed. For example, consider a sequential specification Y for the counter object c described above.

- Suppose that the initial value of the counter is zero. Then

$$\begin{aligned} \langle c\ \mathit{inc}\ A\rangle\langle c\ \mathit{ok}\ A\rangle\langle c\ \mathit{get}\ B\rangle\langle c\ \mathit{ok}(1)\ B\rangle &\quad \in \quad Y \\ \langle c\ \mathit{inc}\ A\rangle\langle c\ \mathit{ok}\ A\rangle\langle c\ \mathit{get}\ B\rangle\langle c\ \mathit{ok}(0)\ B\rangle &\quad \notin \quad Y \end{aligned}$$

- Suppose that the decrement operation blocks if the count is already zero (like a semaphore would). Then Y does not contain any history whose first event is $\langle c\ \mathit{dec}\ A\rangle$.

A sequential specification Y is *nondeterministic* if it contains two distinct histories $H \neq H'$ whose longest common prefix ends in a call, and *deterministic* otherwise.

2.1.3 Operations

Within a history H, we define an *operation* e to be a pair consisting of an invocation $inv(e)$ and the next matching response $res(e)$ (if present). We let $ops(H)$ be the set of all operations in H. We say that an operation e is *pending* if $inv(e)$ is pending, and complete otherwise. We define o_e, i_e, r_e, and t_e to be the object, invocation, response, and thread of the operation e, respectively. We sometimes write down an operation as a bracketed tuple, in the form $[o_e\ \ i_e/r_e\ \ t_e]$ (for complete operations) or $[o_e\ \ i_e/.\ \ t_e]$ (for pending operations).

We define the irreflexive partial order $<_H$ on operations of H by requiring that $e_1 <_H e_2$ if and only if $res(e_1)$ precedes $inv(e_2)$ in H. We say two operations $e_1, e_2 \in ops(H)$ are *overlapping* if neither $e_1 <_H e_2$ nor $e_2 <_H e_1$.

2.1.4 Linearizability

The key idea behind linearizability is to compare concurrent histories to serial histories. We call a history S a *serial witness* for a history H if it satisfies

1. S is serial, and
2. $H|t = S|t$ for all threads t, and
3. $<_H \subseteq <_S$.

Intuitively, a serial witness of H is simply a linear arrangement S of all the operations in H (all but the last one of which must be complete) such that the order of two operations is preserved if they are performed by the same thread, or if they do not overlap.

The following definition is equivalent to the original definition of linearizability, restricted to single-object histories which are the sole focus of our attention in this paper:[2]

DEFINITION 1 (Linearizability). *A single-object history H is linearizable with respect to a sequential specification Y if H can be extended, by appending zero or more return events, to a history H' such that Y contains a serial witness for $complete(H')$.*

2.2 Implementation vs. Specification

Linearizability allows us to specify the desired behavior of a concurrent data type in terms of a sequential specification. It thus gives us the ability to decide whether a particular implementation behaves correctly. As it is well known that the implementation of concurrent data types is difficult (in particular in the presence of performance optimizations or ambitious guarantees such as lock- or wait-freedom), a technique for detecting deviations from the specification is invaluable.

Let X be an implementation of some object. Then we define the set $H(X)$ to be the set of single-object histories exhibited by X, for all possible concurrent programs that may make use of the object, and according to the semantics of the programming platform used by the implementation. We consider X to be linearizable with respect to the specification Y if and only if all histories in $H(X)$ are linearizable with respect to Y.

2.2.1 Example : Buggy Counter 1

We now illustrate how mistakes in an implementation can manifest as linearizability failures. If called concurrently, the buggy counter implementation below (on the left) may exhibit the history H (on the right), because the inc operation fails to acquire a lock:

```
class Counter1
{
  int count = 0;
  void inc() { count = count + 1; }
  int get() { return count; }
  ...
}
```

$\langle c\ \textit{inc}\ A\rangle$
$\langle c\ \textit{inc}\ B\rangle$
$\langle c\ \textit{ok}\ A\rangle$
$\langle c\ \textit{ok}\ B\rangle$
$\langle c\ \textit{get}\ A\rangle$
$\langle c\ \textit{ok}(1)\ A\rangle$

This bug can be detected by a method that checks linearizability, because the history H is not linearizable with respect to the specification in Fig. 3. To see why, consider Def. 1. Because H is already complete, any extension H' as described in the definition must satisfy $H = H' = complete(H')$. Any serial witness S would have to contain exactly the three operations

$e_1 = [c\ \textit{inc}/\textit{ok}\ A] \quad e_2 = [c\ \textit{inc}/\textit{ok}\ B] \quad e_3 = [c\ \textit{get}/\textit{ok}(1)\ A]$

[2] Theorem 1 [15] proves that linearizability of multi-object histories can be soundly reduced to linearizability of single-object histories.

```
class Counter2
{
  int count = 0;
  Lock lock = new Lock();
  void inc() {
    lock.acquire();
    count = count + 1;
    lock.release();
  }
  void get() {
    lock.acquire();
    return count;
  }
  ...
}
```

$\langle c\ \textit{inc}\ A\rangle$
$\langle c\ \textit{ok}\ A\rangle$
$\langle c\ \textit{get}\ A\rangle$
$\langle c\ \textit{ok}(1)\ A\rangle$
$\langle c\ \textit{inc}\ B\rangle$
$\#$

Figure 4. Buggy counter implementation with a stuck history.

and would have to satisfy $e_1 <_S e_3$ and $e_2 <_S e_3$. However, no such serial history is compatible with the specification: if both increment operations precede the get operation, the latter has to return the value 2.

2.2.2 Example : Buggy Counter 2

We now look at a second example of a faulty counter implementation in Figure 4; however, this time the standard linearizability definition fails to detect the bug. If called concurrently, the following buggy counter implementation (left) may exhibit the "stuck" history H (right) because the get operation fails to release the lock:

However, the history H on the right is perfectly linearizable according to Def. 1: adding no return events, we have $H' = H$, and the sequence $\langle c\ \textit{inc}\ A\rangle\langle c\ \textit{ok}\ A\rangle\langle c\ \textit{get}\ A\rangle\langle c\ \textit{ok}(1)\ A\rangle$ is a serial witness for $complete(H')$ (in fact, it is identical to $complete(H')$).

All histories produced by this buggy implementation are linearizable in the sense of Def. 1. This is because Def. 1 only considers whether the values returned by the operations are consistent with the sequential specification, but not whether the operations return in the first place.

Knowing the specification (Figure 3), we can clearly tell that the implementation is never supposed to block inside the operation inc: the only time we would expect it to block is during calls to dec, and only if the current count is 0. We extend the definition of linearizability below so that it can precisely compare the blocking behaviors of the implementation and the specification.

2.3 Generalizing Linearizability

To be able to detect progress problems such as deadlocks in the implementation, we introduce the notion of "stuck" histories. First, we formally define stuck histories to be finite sequences of events ending with the special symbol $\#$. We use the same terminology for stuck histories that we defined for histories (such as complete versus incomplete histories, pending calls, matching returns, and so on). Also, our definition of determinism extends easily to sets of serial histories that contain both regular and stuck histories: a set of serial histories is *nondeterministic* if it contains two histories $H \neq H'$ whose longest common prefix ends with a call.

For an implementation X, we define $\overline{H}(X)$ be the set of stuck histories that the implementation X can exhibit, that is, all histories $\overline{H}$ where (1) $\overline{H}$ has at least one pending operation, and (2) none of the pending operations in $\overline{H}$ can complete due to some inability of the implementation to make progress (such as deadlock, livelock, or a diverging loop). Figure 4 shows an example of a stuck history.

For a stuck history $\overline{H}$ of X to be linearizable, we expect that we can find a stuck serial witness for *all* incomplete operations of $\overline{H}$. This represents the insight that all of the pending operations in the stuck history need to have a justification for being stuck. More formally, for a stuck history $\overline{H}$ and a pending operation e of $\overline{H}$, let $\overline{H}[e]$ be the stuck history obtained from H by removing all pending calls except $inv(e)$. Then we define:

DEFINITION 2 (Linearizability of Stuck Histories). *A stuck single-object history $\overline{H}$ is linearizable with respect to a set Z of stuck serial histories if for each pending operation e of $\overline{H}$, the set Z contains a serial witness for $\overline{H}[e]$.*

Given a sequential specification Y for object o, we define the set $\overline{Y}$ to consist of all histories of the form $H\langle o\ i\ t\rangle\#$ where $H \in Y$ is complete, $i \in I_o$, $t \in T$, and such that there exists no response event x satisfying $H\langle o\ i\ t\rangle x \in Y$. For example, if Y is the specification of the counter object from Section 2.1.2, then $\overline{Y}$ contains (among others) the stuck history $\langle c\ \textit{dec}\ A\rangle\#$.

Finally, we combine the previous two definitions to obtain the general definition of linearizability:

DEFINITION 3 (General Linearizability). *An implementation X is linearizable with respect to a sequential specification Y if all histories in $H(X)$ are linearizable with respect to Y and all stuck histories in $\overline{H}(X)$ are linearizable with respect to $\overline{Y}$.*

2.4 Deterministic Linearizability

We are now ready to formally define the property that Line-Up is checking.

DEFINITION 4 (Deterministic Linearizability). *An implementation X is* deterministically linearizable *if there exists a deterministic sequential specification Y such that X is linearizable with respect to Y.*

Checking deterministic linearizability is useful for finding concurrency-related errors in the implementation. We provide more evidence for this claim in Section 5. Determining whether an implementation is deterministically linearizable is undecidable in general (we assume implementations use a Turing-complete programming language), but useful partial algorithms are nevertheless possible and sensible, as demonstrated in the remainder of this paper.

3. Line-Up Algorithm

We now present our approach to automatically check the deterministic linearizability of an implementation X (for some object o, which remains fixed throughout this section). For better readability, we have moved the detailed proofs to the appendix, and include only short proof descriptions here.

3.1 Tests

A *finite test* for object o is a finite collection of invocations organized by thread. More formally, define a finite test m to be a map $m : T \rightarrow I_o^*$ from threads to invocation sequences, such that $m(t)$ is empty for all but finitely many threads t. We say a finite test m is a prefix of a finite test m' if $m(t)$ is a prefix of $m'(t)$ for all $t \in T$. We sometimes think of finite tests as matrices, with each thread corresponding to a column, and use the corresponding notation, such as

$$m = \begin{bmatrix} \textit{inc} & \textit{get} \\ \textit{inc} & \textit{set}(0) \end{bmatrix} \qquad \text{means} \begin{cases} m(A) = \textit{inc inc} \\ m(B) = \textit{get set}(0) \end{cases}$$

For an object o, and a set of invocations $I \subseteq I_o$, we define the set $M^I_{p\times q}$ to consist of all finite tests corresponding to matrices of dimension $p \times q$ with entries in I.

```
1:  procedure Check(X, m) begin
        // Phase 1: enumerate serial executions of test $m$
2:      $A \leftarrow \hat{M}_s(X, m)$    // all serial full histories
3:      $B \leftarrow \overline{M}_s(X, m)$    // all serial stuck histories
4:      if $A \cup B$ is nondeterministic then
5:          return FAIL
6:      end if
        // Phase 2: check concurrent executions of test $m$
        // check each full history
7:      for all $H \in \hat{M}(X, m)$ do
8:          if $H$ not linearizable with respect to $A$ then
9:              return FAIL
10:         end if
11:     end for
        // check each stuck history
12:     for all $\overline{H} \in \overline{M}(X, m)$ do
13:         if $\overline{H}$ not linearizable with respect to $B$ then
14:             return FAIL
15:         end if
16:     end for
17:     return PASS
18: end
```

Figure 5. The function *Check*(X, m).

3.2 Stateless Model Checker

The significance of using a finite test is that we can employ a stateless model checker to explore all thread schedules the implementation can exhibit for the given finite test. During this enumeration, we instruction the model checker to record the call and return events along with the values of the arguments and return values to generate the history for each execution. We describe this in more depth in Section 4.

For a given test m, we call a history *full* if it is complete and contains all operations of m. For an implementation X and a finite test m, we define $\hat{M}(X, m)$ to be the set of full histories of m found by the model checker, and $\overline{M}(X, m)$ to be the set of stuck histories found by the model checker.

We can also instruct the model checker to explore serial schedules only, if so desired. We define $\hat{M}_s(X, m)$ to be the set of full serial histories found by the model checker, and $\overline{M}_s(X, m)$ to be the set of stuck serial histories found by the model checker.

3.3 The Two-Phase Check

At the core of our method is the function *Check*(X, m) (see Fig. 5). This function checks for a given implementation X and finite test m whether the executions of X for m are consistent with some (unknown) sequential deterministic specification.

The check has two phases. Because we do not know the specification, we *synthesize* it in phase 1, by recording all serial histories of the finite test m. In phase 2, we check whether all concurrent executions are consistent with the specification recorded in phase 1.

The following two theorems describe precisely under what circumstances our test fails or succeeds. The detailed proofs are included in the appendix.

THEOREM 5 (Completeness). *Let X be an implementation and let m be an arbitrary finite test. If Check(X, m) returns FAIL, then X is not deterministically linearizable.*

The theorem holds because if X is linearizable with respect to some deterministic specification, phase 1 is guaranteed to synthesize exactly that specification with respect to m, so we can perform a precise check in phase 2. The guarantee made by this theorem

```
1:  procedure AutoCheck($X$) begin
2:      $n \leftarrow 1$
3:      loop
4:          for all $m \in M^{I_n}_{n \times n}$ do
5:              if Check($X, m$) returns FAIL then
6:                  return FAIL
7:              end if
8:          end for
9:          $n \leftarrow n + 1$
10:     end loop
11: end
```

Figure 6. The algorithm *AutoCheck*(X).

is very strong: it shows that a failing check never produces false alarms, but truly refutes deterministic linearizability.

On the other hand, a return of PASS does not conclusively prove that an implementation is deterministically linearizable, because the chosen finite test may not expose the bug. However, this is the only limitation, as the following theorem demonstrates:

THEOREM 6 (Restricted Soundness). *Let X be an implementation that is not deterministically linearizable. Then there exists a finite test m such that Check(X, m) returns FAIL.*

The theorem holds because if all tests m pass (note that there are infinitely many), then we can construct a deterministic sequential specification for the implementation from all the sets A, B.

In practice, our experience (which we discuss in more depth in Section 5) shows that even this restricted soundness is quite powerful. The reason is that if there exists a finite test that finds the bug, then there usually exists a relatively small one (an empirical observation called "the small scope hypothesis" by some researchers).

3.4 Automatic Algorithm

The checking function *Check*(X, m) still requires a test case m. To achieve full automation, we can generate m automatically. First, determine some enumeration $I_o = \{i_1, i_2, \dots\}$ of the invocations of the object o under test. This can be automatically constructed by enumerating the interface methods of the object and enumerating possible values for each of the methods. For $n \in \mathbb{N}$, define the set $I_n = \{i_1, \dots, i_n\}$ containing the first n elements of I_o. Then we can apply the algorithm *AutoCheck*(X) as shown in Fig. 6 to check linearizability automatically. Note that this algorithm does not terminate on a correct implementation.[3]

Completeness (Thm. 5) holds for *AutoCheck*(X) just as for *Check*(X, m). Soundness is more comprehensive:

THEOREM 7 (Soundness). *Let X be an implementation that is not deterministically linearizable. Then AutoCheck(X) returns FAIL.*

This theorem holds because we know by Thm. 6 that there exists a finite test m such that *Check*(X, m) returns FAIL. We can then choose a sufficiently large n such that m is a prefix of some finite test in $M^{I_n}_{n \times n}$, and apply the following lemma:

LEMMA 8. *If test m is a prefix of test m' and Check(X, m) returns FAIL, then Check(X, m') returns FAIL also.*

Intuitively, the proof works by observing that all full histories of m appear as prefixes in some full or stuck history of m', and all stuck histories of m appear as stuck histories of m'. Note that condition 3 in the definition of a serial witness (Section 2.1.4) is essential for this lemma to hold.

[3] This is consistent with the fact that there cannot be an algorithm for an undecidable problem that is simultaneously sound, complete, and terminating.

4. Implementation

Line-Up is built on top of the stateless model checker CHESS [22], which provides us with the capability of enumerating thread schedules of C# code. We treat the algorithm used by CHESS (fair stateless model checking [21]) and its optimizations and heuristics (e.g. search prioritization [5]) essentially as a black box, thus our algorithm could be used with other model checkers as well. However, we rely on the ability to enumerate schedules *exhaustively*, so simple runtime monitoring is not sufficient. Also, support for fairness is important because many of the concurrent data types use spin-loops for synchronization.

4.1 Implementing *Check*

We implement the two phases of the function *Check*(X, m) (Fig.5) as two separate invocations of CHESS. To perform phase 1, we record all complete and all stuck serial histories. All these histories can be enumerated without preempting threads inside operations, so we instruct CHESS accordingly. The set of observed serial histories Z is recorded in a file (called the *observation file*). To perform phase 2, we run CHESS again, this time exploring the fine-grained thread interleavings, and for each stuck or complete history, we check whether the observation file contains the required serial witness(es). If the check fails, we report the violating history to the user.

4.2 Observation File Format

We chose an XML-based format for listing a set of observations. This format groups all histories into sections, where all histories in a section exhibit the same operation sequences for each thread (Fig. 7, middle). This format has two advantages. For one, when our algorithm is looking for a serial witness in the observation set, it is enough to search one group, because any serial witness must perform matching operation sequences in each thread. Second, this format is easier to understand and navigate manually if the histories become large.

When we report a linearizability violation to the user we include the violating history (Fig. 7, bottom). Often, the first step in analyzing such a report is to examine the observation file for a clue to why it does not contain a serial witness.

4.3 Random Sampling

We found that a literal implementation of the algorithm *AutoCheck* (Fig. 6) does not perform well in practice. The reason is that (1) the model checker performance starts to drop dramatically when going beyond 3x3 matrices, and (2) using the invocation enumeration sets I_n may require unnaturally large values of n for the right combination of invocations to show up in the test. We thus developed a random sampling technique that performed quite well in practice.

Specifically, our adapted algorithm lets the user provide a list of representative invocations I, the desired dimension $n \times m$ of the matrix, and a sample size k. We then run *Check* on a uniform random sample of k tests from $M^{I}_{n \times m}$. Like *Check* and *AutoCheck*, the function *RandomCheck* is complete, but we no longer have a soundness guarantee (bugs may be missed). However, our empirical results in Section 5 show that random sampling is quite efficient at discovering failing testcases.

Another big practical benefit of random sampling is that it is embarrassingly parallel: it is very easy to distribute the various tests and let each core run *Check* independently.

We also allow users to specify entire sequences of invocations to be used when constructing tests, as well as initial and final sequences of operations to perform before and after each test, respectively. Any professional experience of the tester about how to construct effective tests can thus be easily integrated with the

Thread A	Thread B
Add(200)	*Take*()
Add(400)	*TryTake* ()

```
<observationset>
  <observation>
    <thread id="A">1 2</thread>
    <thread id="B">3 4</thread>
    <op id="1" name="Add">value="200"</op>
    <op id="2" name="Add">value="400"</op>
    <op id="3" name="Take">result="200"</op>
    <op id="4" name="TryTake">result="Fail"</op>
    <history>1[ ]1 3[ ]3 4[ ]4 2[ ]2</history>
  </observation>
  <observation>
    <thread id="A">1 2</thread>
    <thread id="B">3 4</thread>
    <op id="1" name="Add">value="200"</op>
    <op id="2" name="Add">value="400"</op>
    <op id="3" name="Take">result="200"</op>
    <op id="4" name="TryTake">result="400"</op>
    <history>1[ ]1 2[ ]2 3[ ]3 4[ ]4</history>
    <history>1[ ]1 3[ ]3 2[ ]2 4[ ]4</history>
  </observation>
  <observation>
    <thread id="A"></thread>
    <thread id="B">1B</thread>
    <op id="1" name="Take" />
    <history>1[ #</history>
  </observation>
</observationset>
```

```
Line-Up encountered a non-linearizable history:
  <thread id="A">1 2</thread>
  <thread id="B">3 4</thread>
  <op id="1" name="Add">value="200"</op>
  <op id="2" name="Add">value="400"</op>
  <op id="3" name="Take">result="200"</op>
  <op id="4" name="TryTake">result="Fail"</op>
  <history>1[ 3[ ]1 2[ ]3 ]2 4[ ]4</history>
```

Figure 7. (Top) An example 2x2 test for a concurrent FIFO queue implementation. *Add* adds an element to the queue, *TryTake* removes and returns the oldest element or fails if the queue is empty, and *Take* removes and returns the oldest element or blocks if the queue is empty. **(Middle)** An example of what the observation file looks like for this test. The histories are grouped into sections (each `<observation>` element is one such section). All histories in a section agree on the sequence of operations performed by each individual thread (`<thread>` elements show the sequence of operations by each thread, while `<op>` elements show the details of each operation), but differ in the precise interleaving of the operation calls and returns (`<history>` elements show the precise interleaving of each history, in the form `i[` and `]i` for call and return of an operation `i` , respectively). Blocking operations are marked with a letter B, and stuck histories are marked with a final #. **(Bottom)** Example of a linearizability violation report for this test and observation file. The test failed because no serial witness was found: there is only one history in the matching `<observation>` element of the file, and it does not order the call of *TryTake* after the return of *Add*(400), so it is not a serial witness for this observation.

```
procedure RandomCheck$(X, I, i, j, n)$ begin
    $M \leftarrow$ random sample of size $n$ drawn from $M^I_{i\times j}$
    for all $m \in M$ do
        if Check$(X, m)$ returns FAIL then
            return FAIL
        end if
    end for
    return PASS
end
```

Figure 8. The algorithm *RandomCheck*(X).

automatic test generation . Also, the user is always free to specify test matrices directly, a useful feature for testing very specific scenarios or for writing regression tests.

To allow the model checker CHESS to complete its schedule exploration, we found it necessary to use the preemption bounding heuristic in CHESS, thus further compromising on soundness. However, we use no bounding during phase 1, so we can retain the important completeness guarantee (that any reported violation is correct).

5. Results

We now present the practical experiences we gathered when applying Line-Up to 13 classes with a total of 90 methods in the .NET Framework 4.0 concurrency classes (Table 1). We used two different versions of these classes, a technology preview (indicated by Pre in our results) and the beta2 release of the library. The two versions are separated by more than a year of development and much of the implementation and the APIs were changed between these two versions. Some of the bugs in the older version were originally found by (an earlier version of) Line-Up and have been fixed in the latest version.

5.1 Approach

For each class, we run the *RandomCheck* procedure to test a uniform random sample of 100 tests of dimension 3×3. To track down the observed failures, we manually remove operations from failing 3x3 test matrices to obtain a failing test of minimal dimension, for the sake of easier reasoning and regression testing. We then analyze the reduced failures further to determine the root cause of the failure.

Table 2 summarizes our results. Each line corresponds to a class we tested (Table 1). A class with the suffix (Pre) indicates a version from the older release of the .NET Framework 4.0.

5.2 Line-Up Failures

For each failure reported by Line-Up, we identified 12 unique root causes indicated by **A** to **L** in Table 2. The table also shows the minimum dimension of a test that is sufficient to find each root cause. The numbers show that most failures can be found with very small tests, confirming the small scope hypothesis.

A Line-Up failure indicates that the class is not linearizable with respect to any deterministic sequential specification. Accordingly, the root cause can be classified as one of the following three categories; we describe each cateogory in a separate subsection.

1. **A Bug** (7 found): The class is intended to be deterministically linearizable with respect to the set of methods tested. Thus the failure indicates a bug in the program.
2. **Intentional Nondeterminism** (3 found): The tested methods are intended to be linearizable but with respect to a nondeterministic specification.

Class	LOC	Properties & Methods checked
Lazy Initialization	470	Value, ToString, IsValueCreated
ManualResetEvent	868	Set, Wait, Reset, IsSet, WaitOne
SemaphoreSlim	585	CurrentCount, Release, Release(2), Wait, Wait(0)
CountdownEvent	571	IsSet, Wait, Wait(0), CurrentCount, WaitOne for x in (ϵ,2) {Signal(x), AddCount(x), TryAddCount(x)}
ConcurrentDictionary	1833	Count, IsEmpty, Clear, for x in (10,20) {TryAdd(x), TryRemove(x), TryGet(x), get[x], set[x], TryUpdate(x), ContainsKey(x)}
ConcurrentQueue	819	Count, IsEmpty, Enqueue, ToArray, TryDequeue, TryPeek
ConcurrentStack	835	Clear, Count, Push, PushRangeTen, TryPop, TryPopRangeOne, TryPopRangeTwo, TryPopRangeFour, TryPeek, ToArrayOrderBy
ConcurrentLinkedList		Count, AddFirst,AddLast,RemoveFirst, RemoveFirst(out value), RemoveList, RemoveLast(out value)
BlockingCollection	1808	Count, ToArray, TryAdd, TryAdd(1), IsCompleted, IsAddingCompleted, CompleteAdding, Add, Take, TakeWithEnum, TryTake, TryTake(1)
ConcurrentBag	1074	Count, Add(10), Add(20), TryTake, IsEmpty, TryPeek, ToArray
TaskCompletionSource	392	Exception, TrySetCanceled, TrySetException, TrySetResult, SetCanceled, SetException, SetResult, Wait, TryResult
CancellationTokenSource	946	Increment, Cancel
Barrier	870	SignalAndWait, ParticipantsRemaining, RemoveParticipant, CancelToken, CurrentPhaseNumber, ParticipantCount, AddParticipant

Table 1. Classes from .NET Framework 4.0 (beta 2) that we used to evaluate Line-Up. The description of these classes is available at [20].

Class	Root Causes			Smallest	Phase 1				Phase 2				
	Bugs	Intentional		failing	#histories		time [min]		#	#	avg. time	avg. time	
		nondet.	nonlin.	testcase	avg	max	avg	max	pass	fail	pass [min]	fail [min]	PB
Lazy Initialization					60	68	0.07	0.07	100	0	9.7		2
ManualResetEvent(Pre)	A			A:2x3	828	1680	3.31	8.36	93	7	12.9	19.9	2
ManualResetEvent					76	133	0.07	0.07	100	0	170.5		2
SemaphoreSlim					1511	1680	3.91	7.31	100	0	18.1		1
CountdownEvent					410	1680	1.20	4.95	100	0	6.3		2
ConcurrentDictionary					1678	1680	7.02	10.94	100	0	6.8		1
ConcurrentQueue(Pre)	B			B:2x2	1680	1680	6.00	6.33	43	57	33.8	5.7	2
ConcurrentQueue					1680	1680	6.37	6.77	100	0	33.7		2
ConcurrentLinkedList(Pre)	C			C:2x2	1674	1680	8.54	12.36	80	20	5.8	2.0	1
ConcurrentStack					1680	1680	3.49	3.68	100	0	1.3		2
BlockingCollection(Pre)	D		K	D:2x2 K:2x2	850	1680	4.09	17.84	13	87	109.6	47.4	2
BlockingCollection		H, I	K	H: 2x2 I:3x2	259	501	0.07	0.07	43	57	205.3	68.7	2
ConcurrentBag(Pre)	E	J		E:2x2	1680	1680	4.18	4.52	75	25	1.7	0.4	1
ConcurrentBag		J		J: 2x2	1680	1680	4.05	4.43	87	13	0.6	3.2	2
TaskCompletionSource(Pre)	F			F:2x2	1432	1680	7.30	20.26	65	35	32.9	9.4	2
TaskCompletionSource	G			G:2x2	1209	1680	5.98	11.95	53	47	53.3	3.7	2
CancellationTokenSource					1680	1680	4.49	5.74	100	0	9.6		2
Barrier			L	L:2x2	1166	1680	4.10	7.02	22	78	7.0	1.5	2

Table 2. Results of applying Line-Up. to classes from the .NET Framework 4.0. The classes from the Parallel Extensions preview [19] are marked with (Pre), all others are from the Beta 2 release.

3. **Intentional Nonlinearizability** (2 found): The tested methods are not intended to be linearizable.

5.2.1 Bugs

Line-Up found a total of 7 bugs tagged **A** to **G** in Table 2. These bugs range from simple synchronization errors to flaws in the logic of the algorithm. In many cases, understanding the bugs requires an intimate knowledge of the class implementation.

We describe one bug (**A**) in the ManualResetEvent class in more detail. This class implements an event, and as one would expect, a thread waits till the event is set by another thread. The event can also be reset after being set. For the test shown in Figure 9, Line-Up produced a concurrent execution in which Thread 1 was never unblocked. We can be convince ourselves that this is an erroneous (or at least, an unexpected) behavior even without understanding how this class is implemented. On careful inspection, we identified the source of the bug to an erroneous use of the compare and swap (CAS) operation shown below:

```
int state;
Wait(){
  //...
  int localstate = state;
  int newstate = f(state); // compute new value
  compare_and_swap(&state, localstate, newstate);
  //...
}
```

Thread 1	Thread 2
mre.Wait();	*mre.Set*();
	mre.Reset();
	mre.Set();

Figure 9. A ManualResetEvent test. Irrespective of the interleaving between the two threads, one expects Thread 1 to be eventually unblocked.

It is common for concurrent algorithms to use a CAS operation to atomically update a shared variable. The correct usage is to read the shared variable into a local copy, use this copy to compute the new value of the variable, and update the shared variable only if the variable has not been modified by another thread in the interim. However, the implementation above contains a pernicious typographical error where the shared variable `state` is read the second time when computing the new value. Arguably, this bug is very hard to find by manual inspection. Moreover, even when the bug is known, it is very hard to design a test harness that exposes the bug: the value of `state` needs to change between the two reads but needs to be set to the first value before the CAS operation, which would otherwise fail.

5.2.2 Intentional Nondeterminism

Line-Up reported 3 failures (**H**, **I**, **J**) that resulted from nondeterminism in the specification. A ConcurrentBag represents an unordered collection of items and the implementation is allowed to remove any one of the elements during a `TryTake`. For the BlockingCollection, Line-Up generated a test where the `Count` method could return 0 even when the collection is not empty. Similarly, it generated a test where the `TryTake` method can fail even when the collection is not empty. While these are clearly unexpected behaviors, the developers of these classes were unable to provide a fix that was both efficient and did not require global changes to the implementation, and decided instead to change the official documentation of these methods to include the potentially nondeterministic behavior.

5.3 Intentional Nonlinearizability

Line-Up reported 2 instances (**K**, **L**) of nonlinearizability. The BlockingCollection contains a `Cancel` method where the effects of cancellation can take place well after the method has returned. This class is linearizable for all other methods. The Barrier is a classic example of a nonlinearizable class. Barriers block each thread until all threads have entered the barrier, a behavior that is not equivalent to any serial execution.

In summary, the results above show that a large variety of concurrency errors can be caught quite easily as a linearizability violation of randomly chosen 3x3 tests. They also show that there is some room for improvement in extending our method to support nondeterministic or nonlinearizable methods.

5.4 Runtime of the Two Phases

The table 2 shows the runtime characteristics of the two phases. For Phase 1, the enumeration of serial histories, the table shows how many histories were observed on average and on maximum. The table also shows how long it takes to run phase 1 on a single 3x3 testcase (average and maximum observed). All measurements were performed on an 8 Core 2.33 GHz Intel Xeon, with one core assigned to one testcase at a time. The numbers demonstrate that the automatic enumeration of a sequential specification is very cheap, which is a key fact exploited by the Line-Up algorithm.

For Phase 2, the enumeration of concurrent histories, the table shows how many testcases passed and failed. Most violations were caught by a large proportion of the sample. We also show how long it takes to complete a failing/passing testcase on average. As usual, testcases fail much quicker than they pass. The PB column shows the preemption bound used to limit the search. We use 2 (the CHESS default) except where it performed unacceptably slow.

5.5 Relevance of using generalized linearizability

Because we are generating random tests, some tests may get stuck. For example, a random test may easily acquire a blocking semaphore more often than release it, thus causing deadlock (in both phases). This is reflected by the fact that the number of histories enumerated in phase one is sometimes less than the combinatorial number of full histories for 3x3 matrices, which is 1680.

Our use of generalized linearizability (as opposed to classical linearizability) is significant insofar 5 of the 13 classes tested exhibited deadlocking tests and could not have been tested with a methodology that can not handle them. In particular, we would not be able to single out the bug in Figure 9 with a tool that checks standard (nonblocking) linearizability only.

5.6 Comparisons

To compare Line-Up with other dynamic checking methods, we also checked for data races and atomicity violations. For data race detection, we used the happens-before based dynamic race detector included with CHESS. To perform atomicity checking, we implemented the algorithm described in [10], which checks whether a given dynamic execution is conflict-serializable.

All the data races we found were benign, and it appeared that they remained in the code only because of limitations in the C# compiler (which does not permit arrays of volatiles, or references to volatiles). We believe the surprisingly low number of data races we found shows that, for this type of application, a conservative use of volatile declarations and interlocked operations is a relatively simple way to avoid data races (but does of course nothing to prohibit higher-level mistakes in the logic of the algorithm).

Our experiments with atomicity (conflict serializability) checking resulted in hundreds of warnings. We abandoned the effort of classifying these warnings into real errors after inspecting the first ten of them which turned out to be false alarms. Our initial investigation yielded four good and common reasons why programs (in particular, concurrent data types) can exhibit non-serializable executions and still be correct.[4] We briefly list these here.

1. (ConcurrentStack, ConcurrentQueue) The code performs a CAS. A failing CAS leads to a retry. However, the accesses performed before the retry break serializability.
2. (SemaphoreSlim) The code contains a timing optimization (similar to double-checked locking) that does not affect correctness, but breaks serializability.
3. (CancellationTokenSource) The current state is read and compared using a $\geq$ operator. At an abstract level, this comparison is a right-mover, but a simple serializability detector does not know that.
4. (ConcurrentBag) A thread performs lazy initialization, acquiring a global lock. This work does not affect the current operation in any way, but breaks serializability.

It is labor-intensive to decide whether an atomicity violation is benign as doing so requires a good understanding of the design principles of the implementation, and we did not write this code our-

[4] We believe that all of the observed non-serializable behaviors could be made serializable by exploiting global knowledge about invariants and commutativity of operations (e.g. using atomic blocks and movers [7]), but automating such an approach would indeed be very challenging.

selves. Using Line-Up was easier because the reported violations of linearizability provided a sufficiently conclusive evidence of malfunction to convince the original developers that they had to analyze and fix the problem.

5.7 Memory model issues

The algorithm as presented in this paper is orthogonal to memory model issues in the sense that it is the responsibility of the underlying model checker to produce all histories (including histories that manifest for non-sequentially consistent executions). However, the CHESS model checker does not directly enumerate the relaxed behaviors of the target architecture; instead it checks for potential violations of sequential consistency using a special algorithm similar to data race detection [3]. We thus used this technique, but did not find any such issues in the studied implementations.

6. Conclusion and Future Work

We have made several important contributions in this paper. First, we show how to improve the definition of linearizability so it can detect erroneous blocking in implementations. Then, we present an automatic algorithm to detect linearizability violations, and show how it can be practically implemented and applied using a stateless model checker.

We then demonstrated on real production code that our tool Line-Up can detect a variety of concurrency bugs automatically. Our work shows the practical value of a tool that checks implementations without requiring knowledge of their design principles, and the appeal of error reports that show a specific scenario where the component misbehaves in an externally observable way.

As future work, we would like to take Line-Up beyond simple linearizable methods and incorporate support for (1) asynchronous methods, such as the cancel method, and (2) nondeterministic methods, such as methods that may fail on interference.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2), 1991.

[2] S. Burckhardt, R. Alur, and M. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *Programming Language Design and Impl. (PLDI)*, pages 12–21, 2007.

[3] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *Computer-Aided Verification (CAV)*, pages 107–120, 2008.

[4] R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set algorithm. In *Computer-Aided Verification (CAV)*, LNCS 4144, pages 475–488. Springer, 2006.

[5] K. Coons, M. Musuvathi, and S. Burckhardt. Gambit: Effective unit testing of concurrency libraries. In *Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[6] S. Doherty, D. Detlefs, L. Grove, C. Flood, V. Luchangco, P. Martin, M. Moir, N. Shavit, and G. Steele. DCAS is not a silver bullet for nonblocking algorithm design. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 216–224, 2004.

[7] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *Principles of Programming Languages (POPL)*, 2009.

[8] T. Elmas and S. Tasiran. VyrdMC: Driving runtime refinement checking with model checkers. *Electr. Notes Theor. Comput. Sci.*, 144:41–56, 2006.

[9] T. Elmas, S. Tasiran, and S. Qadeer. VYRD: verifying concurrent programs by runtime refinement-violation detection. In *Programming Language Design and Impl. (PLDI)*, pages 27–37, 2005.

[10] A. Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *Computer-Aided Verification (CAV)*, 2008.

[11] C. Flanagan and S. Freund. Efficient and precise dynamic race detection. In *Programming Language Design and Impl. (PLDI)*, 2009.

[12] C. Flanagan, S. Freund, and J.Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *Programming Language Design and Impl. (PLDI)*, 2008.

[13] K. Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, 2004.

[14] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), 2007.

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

[16] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comp.*, C-28(9):690–691, 1979.

[17] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective sampling for lightweight data-race detection. In *Programming Language Design and Impl. (PLDI)*, 2009.

[18] M. Michael and M. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, 1995.

[19] MSDN, `http://blogs.msdn.com/somasegar/archive/2007/11/29/parallel-extensions-to-the-net-fx-ctp.aspx`. *Parallel Extensions to the .NET FX CTP*, November 2007.

[20] MSDN, `http://msdn.microsoft.com/en-us/library/dd460718(VS.100).aspx`. *.NET Framework 4 Data Structures for Parallel Programming*, November 2009.

[21] M. Musuvathi and S. Qadeer. Fair stateless model checking. In *Programming Language Design and Impl. (PLDI)*, 2008.

[22] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Operating Systems Design and Impl. (OSDI)*, pages 267–280, 2008.

[23] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 4(26), October 1979.

[24] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comp. Sys.*, 15(4):391–411, 1997.

[25] V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*. Springer-Verlag, 2009.

[26] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 129–136, 2006.

[27] M. Vechev, E. Yahav, and G. Yorsh. Experience with model checking linearizability. In *SPIN*, 2009.

A. Proofs

We first establish the following lemma which states that the enumeration done in phase 1 provides all the necessary histories. For each history H, let m_H be the corresponding test (that is, we let $m_H(t)$ be the sequence of calls made by thread t in H).

LEMMA 9 (Specification Synthesis.). *Let X be a linearizable implementation of some object o with respect to a deterministic sequential specification Y.*

1. *If $H \in Y$ is complete, then $H \in \hat{M}_{\mathrm{s}}(X, m_H)$.*
2. *If $H \in \overline{Y}$, then $H \in \overline{M}_{\mathrm{s}}(X, m_H)$.*

PROOF. Let $H \in Y$ or $H \in \overline{Y}$ (thus H is serial). Now consider an execution of the test m_H where the schedule is restricted in such a way that calls are made in the exact sequence that they appear in H, and no call is made before the previous call returns. Such an execution must be possible without getting stuck prematurely (any stuck partial execution would not be consistent with the fact that X is linearizable w.r.t. Y, because that would imply that $H'\#$ is in $\overline{Y}$ for some prefix H' of H which is impossible because H in Y and Y is deterministic) and all the returns must match the ones in the sequence H (because X is linearizable w.r.t. Y and Y is deterministic). Thus, such an execution must end up reproducing H. If H is complete, it is full for m_H, and thus $H \in \hat{M}_{\mathrm{s}}(X, m_H)$. On the other hand, if $H \in \overline{Y}$, then the execution can not possibly continue with a return (such a continuation, since linearizable, would again contradict the determinism of Y). Thus, it must be stuck, and thus $H \in \overline{M}_{\mathrm{s}}(X, m_H)$. $\square$

A.1 Proof of Thm. 5

We assume there exists a deterministic spec Y such that X is linearizable with respect to Y, but assume that *Check*(X, m) nevertheless returns FAIL, and show that a contradiction results. Distinguish cases.

Case 1: The check on line 4 returns FAIL. Then there must exist histories $H \neq H'$ in $A \cup B$ whose maximal common prefix ends in a call. But then both of H, H' must be in Y (or $\overline{Y}$) if they are full (or stuck), because the only serial witness for a serial history is that exact history. But that contradicts the assumption that Y is deterministic or the definition of $\overline{Y}$.

Case 2: The check on line 8 returns FAIL for $H \in \hat{M}(X, m)$. Because X is linearizable and complete, we know H has a serial witness $S \in Y$. By Lemma 9, we know $S \in \hat{M}_{\mathrm{s}}(X, m_H)$. Because H is a full execution of m, we know $m_H = m$, thus $S \in \hat{M}_{\mathrm{s}}(X, m)$. But that implies that $S \in A$ after phase 1 (see Fig. 5) so when the check on line 8 is performed, it does not fail, contradicting our assumption.

Case 3: The check on line 13 returns FAIL for $\overline{H} \in \overline{M}(X, m)$. Let e be the operation in $\overline{H}$ for which $\overline{H}[e]$ fails the linearizability test (in the sense of Def. 2). Because X is linearizable, there exists by Def. 2 a serial witness $S \in \overline{Y}$ for $\overline{H}[e]$. By Lemma 9, we know $S \in \overline{M}_{\mathrm{s}}(X, m_{\overline{H}[e]})$. Now, because $m_{\overline{H}[e]}$ is a prefix of m, this implies $S \in \overline{M}_{\mathrm{s}}(X, m)$. But that implies that $S \in B$ after phase 1, so the check on line 13 could not have failed as we assumed.

A.2 Proof of Thm. 6

Assume that there exists no test m such that *Check*(X, m) returns FAIL. For a test m, let A_m, B_m be the sets computed in *Check*(X, m), and let B'_m be the set of histories obtained from B_m by removing all pending calls and symbols $\#$. Then define Y to be the prefix-closure of $\bigcup_m (A_m \cup B'_m)$ and observe the following:

1. Y is deterministic. If not, it would contain distinct histories whose longest common prefix ends with a call, but which continue differently (with different returns). Tracking these back to the tests m, m' where they came from, we can reason that the model checker would produce the same options to continue this same prefix in both *Check*(X, m) and *Check*(X, m'), which implies that the nondeterminism would be detected on the lines 4, contrary to the assumption that all tests pass.
2. If $\overline{H} \in B_m$, then $\overline{H} \in \overline{Y}$. To see this, let $\overline{H} = H\langle o\ i\ t\rangle\#$; then $H \in Y$. For the same reason as argued in observation 1, H can not be the prefix of any longer history in Y, thus $\overline{H}$ in $\overline{Y}$.
3. Each element of $\hat{M}(X, m)$ is linearizable with respect to Y: the check on line 8 passed, which implies that A_m contains a serial witness, thus Y does too.
4. Each element of $\overline{M}(X, m)$ is linearizable with respect to $\overline{Y}$: the check on line 13 passed, which implies that B_m contains a serial witness for each $\overline{H}[e]$. By observation 2 this implies that $\overline{Y}$ also contains those serial witnesses.

A.3 Proof of Thm. 7

We know by Thm. 6 that there exists a test m such that *Check*(X, m) returns FAIL. Now, pick an n such that (1) I_n contains all the invocations appearing in m, and (2) n is larger than the largest thread such that $m(t)$ is nonempty, and (3) n is larger than length of the longest sequence $m(t)$, for any t. This then implies that $M^{I_n}_{n\times n}$ contains a test m' such that m is a prefix of m'. The claim then follows by Lemma 8.

A.4 Proof of Lemma 8

We assume that the test m is a prefix of a test m', that *Check*(X, m) returns FAIL, that *Check*(X, m') returns PASS, and show that a contradiction results. Let $A = A_m$, $B = B_m$, $A' = A_{m'}$, and $B' = B_{m'}$. Now we distinguish by where the check fails.

Case 1: The check on line 4 returns FAIL. Then there must exist histories $H \neq H'$ in $A \cup B$ whose maximal common prefix ends in a call. Now, because m is a prefix of m', the model checker will explore histories of m' that follow H exactly, but continue (if H is not already stuck) until they are full or stuck. Thus each of H, H' is either in B' (if it is stuck) or is a prefix of some longer history in $A' \cup B'$. This implies that the check on line 4 fails in *Check*(X, m') as well, contradicting the assumption.

Case 2: The check on line 13 returns FAIL for $\overline{H} \in \overline{M}(X, m)$. Let e be the operation in $\overline{H}$ for which $\overline{H}[e]$ fails the linearizability test (in the sense of Def. 2). Since m is a prefix of m', $\overline{H}$ is also a stuck history of m', and since *Check*(X, m') passed, B' must contain a serial witness S for $\overline{H}'[e]$. But because S' can only contain operations that are in m, this implies also that $S' \in B$ which contradicts the assumption that the test on line 13 failed.

Case 3: The check on line 8 returns FAIL for $H \in \hat{M}(X, m)$. Now, because m is a prefix of m', the model checker will explore histories of m' that start with H but continue until they are full or stuck. Let's distinguish these cases. (Case 3a) H is a prefix of a full $H' \in \hat{M}(X, m')$. Then because the check passed for m', there exists a serial witness $S' \in A'$ for H'. Because of condition 3 in the definition of a serial witness (Section 2.1.4), we know that within S', all operations of $ops(H)$ must precede the operations of $ops(H') - ops(H)$. Thus there exists a prefix S of S' that contains exactly the operations $ops(H)$ and that is thus a serial witness for H. But then we would necessarily have $S \in A$ which contradicts the assumption that *Check*(X, m) returns FAIL. (Case 3b) H is a prefix of a stuck $\overline{H} \in \overline{M}(X, m')$. Then because the check passed for m', there exists a serial witness $S' \in B'$ for $\overline{H}[e]$ for some e. Because of condition 3 in the definition of a serial witness (Section 2.1.4), we know that within S', all operations of $ops(H)$ must precede the operations of $ops(H') - ops(H)$. Thus there exists a prefix S of S' that contains exactly the operations $ops(H)$ and that is thus a serial witness for H. But then we would necessarily have $S \in A$ which contradicts the assumption that *Check*(X, m) returns FAIL.