On Correctness of Data Structures under Reads-Write Concurrency *

Kfir Lev-Ari[1], Gregory Chockler[2], and Idit Keidar[1]

[1] EE Department, Technion, Israel
[2] CS Department, Royal Holloway, UK

Abstract. We study the correctness of shared data structures under reads-write concurrency. A popular approach to ensuring correctness of read-only operations in the presence of concurrent update, is read-set validation, which checks that all read variables have not changed since they were first read. In practice, this approach is often too conservative, which adversely affects performance. In this paper, we introduce a new framework for reasoning about correctness of data structures under reads-write concurrency, which replaces validation of the entire read-set with more general criteria. Namely, instead of verifying that all read shared variables still hold the values read from them, we verify abstract conditions over the shared variables, which we call *base conditions*. We show that reading values that satisfy some base condition at every point in time implies correctness of read-only operations executing in parallel with updates. Somewhat surprisingly, the resulting correctness guarantee is not equivalent to linearizability, and is instead captured through two new conditions: *validity* and *regularity*. Roughly speaking, the former requires that a read-only operation never reaches a state unreachable in a sequential execution; the latter generalizes Lamport's notion of regularity for arbitrary data structures, and is weaker than linearizability. We further extend our framework to capture also linearizability. We illustrate how our framework can be applied for reasoning about correctness of a variety of implementations of data structures such as linked lists.

1 Introduction

Motivation Concurrency is an essential aspect of computing nowadays. As part of the paradigm shift towards concurrency, we face a vast amount of legacy sequential code that needs to be parallelized. A key challenge for parallelization is verifying the correctness of the new or transformed code. There is a fundamental tradeoff between generality and performance in state-of-the-art approaches to correct parallelization. General purpose methodologies, such as transactional memory [13, 23] and coarse-grained locking, which do not take into account the

* This work was partially supported by the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI), by the Israeli Ministry of Science, by a Royal Society International Exchanges Grant, and by the Randy L. and Melvin R. Berlin Fellowship in the Cyber Security Research Program.

inner workings of a specific data structure, are out-performed by hand-tailored fine-grained solutions [19]. Yet the latter are notoriously difficult to develop and verify. In this work, we take a step towards mitigating this tradeoff.

It has been observed by many that correctly implementing concurrent modifications of a data structure is extremely hard, and moreover, contention among writers can severely hamper performance [21]. It is therefore not surprising that many approaches do not allow write-write concurrency; these include the *read-copy-update (RCU)* approach [18], flat-combining [12], coarse-grained readers-writer locking [8], and pessimistic software lock-elision [1]. It has been shown that such methodologies can perform better than ones that allow write-write concurrency, both when there are very few updates relative to queries [18] and when writes contend heavily [12]. We focus here on solutions that allow only read-read and read-write concurrency.

A popular approach to ensuring correctness of read-only operations in the presence of concurrent updates, is *read-set validation*, which checks that no shared variables have changed since they were first read. In practice, this approach is often too conservative, which adversely affects performance. For example, when traversing a linked list, it suffices to require that the last read node is connected to the rest of the list; there is no need to verify the values of other traversed nodes, since the operation no longer depends on them. In this paper, we introduce a new framework for reasoning about correctness of concurrent data structures, which replaces validation of the entire read-set with more general conditions: instead of verifying that all read shared variables still hold the values read from them, we verify abstract conditions over the variables. These are captured by our new notion of *base conditions*.

Roughly speaking, a *base condition* of a read-only operation at time t, is a predicate over shared variables, (typically ones read by the operation), that determines the local state the operation has reached at time t. Base conditions are defined over sequential code. Intuitively, they represent invariants the read-only operation relies upon in sequential executions. We show that the operation's correctness in a concurrent execution depends on whether these invariants are preserved by update operations executed concurrently with the read-only one. We capture this formally by requiring each state in every read-only operation to have a *base point* of some base condition, that is, a point in the execution where the base condition holds. In the linked list example – it does not hurt to see old values in one section of the list and new ones in another section, as long as we read every next pointer consistently with the element it points to. Indeed, this is the intuition behind the famous hand-over-hand locking (lock-coupling) approach [20, 3].

Our framework yields a methodology for verifiable reads-write concurrency. In essence, it suffices for programmers to identify base conditions for their sequential data structure's read-only operations. Then, they can transform their sequential code using means such as readers-writer locks or RCU, to ensure that read-only operations have base points when run concurrently with updates.

It is worth noting that there is a degree of freedom in defining base conditions. If coarsely defined, they can constitute the validity of the entire read-set, yielding coarse-grained synchronization as in snapshot isolation and transactional memories. Yet using more precise observations based on the data structure's inner workings can lead to fine-grained base conditions and to better concurrency. Our formalism thus applies to solutions ranging from validation of the entire read-set [9], through multi-versioned concurrency control [5], which has read-only operations read a consistent snapshot of their entire read-set, to fine-grained solutions that hold a small number of locks, like hand-over-hand locking.

Overview of Contributions This paper makes several contributions that arise from our observation regarding the key role of base conditions. We observe that obtaining base points of base conditions guarantees a property we call *validity*, which specifies that a concurrent execution does not reach local states that are not reachable in sequential ones. Intuitively, this property is needed in order to avoid situations like division by zero during the execution of the operation. To incorporate read-time order, we restrict base point locations to ones that follow all operations that precede the read-only operation, and precedes ones that ensue it. Somewhat surprisingly, this does not suffice for the commonly-used correctness criterion of *linearizability (atomicity)* [14] or even *sequential consistency* [15] (discussed in the full paper [17]). Rather, it guarantees a correctness notion weaker than linearizability, similar to Lamport's *regularity* semantics for registers, which we extend here for general objects for the first time.

In Section 2, we present a formal model for shared memory data structure implementations and executions, and define correctness criteria. Section 3 presents our methodology for achieving regularity and validity: We formally define the notion of a base condition, as well as base points, which link the sequentially-defined base conditions to concurrent executions. We assert that base point consistency implies validity, and that the more restricted base point condition, which we call *regularity base point consistency*, implies regularity (formal proofs appear in the full paper). We proceed to exemplify our methodology for a standard linked list implementation, in Section 4 (see the full paper for more examples). In Section 5 we turn to extend the result for linearizability. We define a condition on update operations, namely, having a *single visible mutation point (SVMP)*, which along with regularity base point consistency ensures linearizability.

We note that we see this paper as the first step in an effort to simplify reasoning about fine-grained concurrent implementations. It opens many directions for future research, which we overview in Section 6. Due to space considerations, some formal definitions and proofs are deferred to the full paper, as is our result about sequential consistency.

Comparison with Other Approaches The regularity correctness condition was introduced by Lamport [16] for registers. To the best of our knowledge, the regularity of a data structure as we present in this paper is a new extension of the definition.

Using our methodology, proving correctness relies on defining a base condition for every state in a given sequential implementation. One easy way to do so is to define base conditions that capture the entire read-set, i.e., specify that there is a point in the execution where all shared variables the operation has read hold the values that were first read from them. But often, such a definition of base conditions is too strict, and spuriously excludes correct concurrent executions. Our definition generalizes it and thus allows for more parallelism in implementations.

Opacity [11] defines a sufficient condition for validity and linearizability, but not a necessary one. It requires that every transaction see a consistent snapshot of all values it reads, i.e., that all these values belong to the same sequentially reachable state. We relax the restriction on shared states busing base conditions.

Snapshot isolation [4] guarantees that no operation ever sees updates of concurrent operations. This restriction is a special case of the possible base points that our base point consistency criterion defines, and thus also implies our condition for the entire read-set.

We prove that the SVMP condition along with regularity base point consistency suffices for linearizability. There are mechanisms, for example, transactional memory implementations [9], for which it is easy to see that these conditions hold for base conditions that capture the entire read-set. Thus, the theorems that we prove imply, in particular, correctness of such implementations.

In this paper we focus on correctness conditions that can be used for deriving a correct data structure that allows reads-write concurrency from a sequential implementation. The implementation itself may rely on known techniques such as locking, RCU [18], pessimistic lock-elision [1], or any combinations of those, such as RCU combined with fine-grained locking [2]. There are several techniques, such as flat-combining [12] and read-write locking [8], that can naturally expand such an implementation to support also write-write concurrency by adding synchronization among update operations.

Algorithm designers usually prove linearizability of by identifying a serialization point for every operation, showing the existence of a specific partial ordering of operations [7], or using rely-guarantee reasoning [24]. Our approach simplifies reasoning – all the designer needs to do now is identify a base condition for every state in the existing sequential implementation, and show that it holds under concurrency. This is often easier than finding and proving serialization points, as we exemplify. In essence, we break up the task of proving data structure correctness into a generic part, which we prove once and for all, and a shorter, algorithm-specific part. Given our results, one does not need to prove correctness explicitly (e.g., using linearization points or rely-guarantee reasoning, which typically result in complex proofs). Rather, it suffices to prove the much simpler conditions that read-only operations have base points and updates have an SVMP, and linearizability follows from our theorems. Another approach that simplifies verifiable parallelization is to re-write the data structure using primitives that guarantee linearizability such as LLX and SCX [6]. Whereas the latter focuses on non-blocking concurrent data structure implementations using

their primitive, our work is focused on reads-write concurrency, and does not restrict the implementation; in particular, we target lock-based implementations as well as non-blocking ones.

2 Model and Correctness Definitions

We consider a shared memory model where each process performs a sequence of operations on shared data structures. The data structures are implemented using a set $X = \{x_1, x_2, ...\}$ of shared variables. The shared variables support atomic read and write operations (i.e., are atomic registers), and are used to implement more complex data structures. The values in the x_i's are taken from some domain $\mathcal{V}$.

2.1 Data Structures and Sequential Executions

A *data structure implementation* (algorithm) is defined as follows:

- A set of states, $\mathcal{S}$, were a *shared state* $s \in \mathcal{S}$ is a mapping $s : X \to \mathcal{V}$, assigning values to all shared variables. A set $\mathcal{S}_0 \subseteq \mathcal{S}$ defines *initial states*.
- A set of operations representing methods and their parameters. For example, $find(7)$ is an operation. Each *operation op* is a state machine defined by:
  - A set of local states $\mathcal{L}_{op}$, which are usually given as a set of mappings l of values to local variables. For example, for a local state l, $l(y)$ refers to the value of the local variable y in l. $\mathcal{L}_{op}$ contains a special initial local state $\perp \in \mathcal{L}_{op}$.
  - A deterministic transition function $\tau_{op}(\mathcal{L}_{op} \times \mathcal{S}) \to Steps \times \mathcal{L}_{op} \times \mathcal{S}$ where $step \in Steps$ is a transition label, which can be *invoke*, $a \leftarrow$ *read*(x_i), *write*(x_i,v), or *return*(v) (see the full paper for more details). Note that there are no atomic read-modify-write steps. Invoke and return steps interact with the application while read and write steps interact with the shared memory.

  We assume that every operation has an isolated state machine, which begins executing from local state $\perp$.

For a transition $\tau(l, s) = \langle step, l', s' \rangle$, l determines the step. If *step* is an invoke, return, or write step, then l' is uniquely defined by l. If *step* is a read step, then l' is defined by l and s, specifically, $read(x_i)$ is determined by $s(x_i)$. Since only write steps can change the content of shared variables, $s = s'$ for invoke, return, and read steps.

For the purpose of our discussion, we assume the entire shared memory is statically allocated. This means that every read step is defined for every shared state in $\mathcal{S}$. One can simulate dynamic allocation in this model by writing to new variables that were not previously used. Memory can be freed by writing a special value, e.g., "invalid", to it.

A state consists of a local state l and a shared state s. By a slight abuse of terminology, in the following, we will often omit either shared or local component

of the state if its content is immaterial to the discussion. A *sequential execution of an operation* is an alternating sequence of steps and states with transitions being according to τ. A *sequential execution of a data structure* is a sequence of operation executions that begins in an initial state; see the full paper for a formal definition. A *read-only operation* is an operation that does not perform write steps in any execution. All other operations are *update operations*.

A state is *sequentially reachable* if it is reachable in some sequential execution of a data structure. By definition, every initial state is sequentially reachable. The *post-state* of an invocation of operation o in execution μ is the shared state of the data structure after o's return step in μ; the *pre-state* is the shared state before o's invoke step. Recall that read-only operations do not change the shared state and execution of update operations is serial. Therefore, every pre-state and post-state of an update operation in μ is sequentially reachable. A state st' is sequentially reachable from a state st if there exists a sequential execution fragment that starts at st and ends at st'.

In order to simplify the discussion of initialization, we assume that every execution begins with a dummy (initializing) update operation that does not overlap any other operation.

2.2 Correctness Conditions for Concurrent Data Structures

A *concurrent execution fragment of a data structure* is a sequence of interleaved states and steps of different operations, where state consists of a set of local states $\{l_i, ..., l_j\}$ and a shared state s_k, where every l_i is a local state of a pending operation. A *concurrent execution of a data structure* is a concurrent execution fragment of a data structure that starts from an initial shared state. Note that a sequential execution is a special case of concurrent execution. An example of a concurrent execution is detailed in the full paper.

A *single-writer multiple-reader (SWMR) execution* is one in which update operations are not interleaved; read-only operations may interleave with other read-only operations and with update operations. In the remainder of this paper we discuss only SWMR executions.

For an execution σ of data structure ds, the *history* of σ, denoted H_σ, is the subsequence of σ consisting of the invoke and return steps in σ (with their respective return values). For a history H_σ, *complete(H_σ)* is the subsequence obtained by removing pending operations, i.e., operations with no return step, from H_σ. A history is *sequential* if it begins with an invoke step and consists of an alternating sequence of invoke and return steps.

A data structure's correctness in sequential executions is defined using a *sequential specification*, which is a set of its allowed sequential histories.

Given a correct sequential data structure, we need to address two aspects when defining its correctness in concurrent executions. As observed in the definition of opacity [11] for memory transactions, it is not enough to ensure serialization of completed operations, we must also prevent operations from reaching undefined states along the way. The first aspect relates to the data structure's

external behavior, as reflected in method invocations and responses (i.e., histories):

Linearizability and Regularity A history H_σ is *linearizable* [14] if there exists H'_σ that can be created by adding zero or more return steps to H_σ, and there is a sequential permutation π of complete(H'_σ), such that: (1) π belongs to the sequential specification of ds; and (2) every pair of operations that are not interleaved in σ, appear in the same order in σ and in π. A data structure ds is *linearizable*, also called *atomic*, if for every execution σ of ds, H_σ is linearizable.

We next extend Lamport's regular register definition [16] for SWMR data structures (we do not discuss regularity for MWMR executions, which can be defined similarly to [22]). A data structure ds is *regular* if for every execution σ of ds, and every read-only operation $ro \in H_\sigma$, if we omit all other read-only operations from H_σ, then the resulting history is linearizable.

Validity The second correctness aspect is ruling out bad cases like division by zero or access to uninitialized data. It is formally captured by the following notion of *validity*: A data structure is *valid* if every local state reached in an execution of one of its operations is sequentially reachable. We note that, like opacity, validity is a conservative criterion, which rules out bad behavior without any specific data structure knowledge. A data structure that does not satisfy validity may be correct, but proving that requires care.

3 Base Conditions, Validity and Regularity

3.1 Base Conditions and Base Points

Intuitively, a base condition establishes some link between the local state an operation reaches and shared variables the operation has read before reaching this state. It is given as a predicate Φ over shared variable assignments. Formally:

Definition 1 (Base Condition). Let l be a local state of an operation op. A predicate Φ over shared variables is a *base condition* for l if every sequential execution of op starting from a shared state s such that $\Phi(s) = true$, reaches l.

For completeness, we define a base condition for $step_i$ in an execution μ to be a base condition of the local state that precedes $step_i$ in μ.

Consider a data structure consisting of an array of elements v and a variable $lastPos$, whose last element is read by the function *readLast*. An example of an execution fragment of *readLast* that starts from state s_1 (depicted in Figure 1) and the corresponding base conditions appear in Algorithm 1. The *readLast* operation needs the value it reads from $v[tmp]$ to be consistent with the value of $lastPos$ that it reads into tmp because if $lastPos$ is newer than $v[tmp]$, then $v[tmp]$ may contain garbage. The full paper details base conditions for every possible local state of *readLast*.

(a) s_1 **(b)** s_2

Figure 1: Two shared states satisfying the same base condition $\Phi_3 : lastPos = 1 \wedge v[1] = 7$.

local state	***base condition***	**Function** *readLast*()
l_1 : {}	$\Phi_1 : true$	$tmp \leftarrow$ ***read***$(lastPos)$
l_2 : $\{tmp = 1\}$	$\Phi_2 : lastPos = 1$	$res \leftarrow$ ***read***$(v[tmp])$
l_3 : $\{tmp = 1, res = 7\}$	$\Phi_3 : lastPos = 1 \wedge v[1] = 7$	***return***(res)

Algorithm 1: The local states and base conditions of readLast when executed from s_1. The shared variable $lastPos$ is the index of the last updated value in array v. See Algorithm 2 for corresponding update operations.

The predicate $\Phi_3 : lastPos = 1 \wedge v[1] = 7$ is a base condition of l_3 because l_3 is reachable from any shared state in which $lastPos = 1$ and $v[1] = 7$ (e.g., s_2 in Figure 1), by executing lines 1-2.

We now turn to define base points of base conditions, which link a local state with base condition Φ to a shared state s where $\Phi(s)$ holds.

Definition 2 (Base Point). Let μ be a concurrent execution, ro be a read-only operation executed in μ, and Φ_t be a base condition of the local state and step at index t in μ. An execution fragment of ro in μ has a *base point* for point t with Φ_t, if there exists a sequentially reachable post-state s in μ, called a *base point of* t, such that $\Phi_t(s)$ holds.

Note that together with Definition 1, the existence of a base point s implies that t is reachable from s in all sequential runs starting from s.

We say that a data structure ds satisfies *base point consistency* if every point t in every execution of every read-only operation ro of ds has a base point with some base condition of t.

The possible base points of read-only operation ro are illustrated in Figure 2. To capture real-time order requirements we further restrict base point locations.

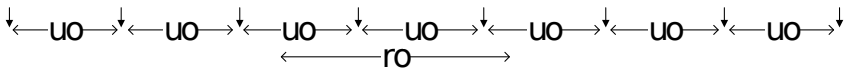


Figure 2: Possible locations of ro's base points.

Definition 3 (Regularity Base Point). A base point s of a point t of ro in a concurrent execution μ is a *regularity base point* if s is the post-state of either an update operation executed concurrently with ro in μ or of the last update operation that ended before ro's invoke step in μ.

The possible regularity base points of a read-only operation are illustrated in Figure 3. We say that a data structure *ds* satisfies *regularity base point consistency* if it satisfies base point consistency, and every return step t in every execution of every read-only operation *ro* of *ds* has a regularity base point with a base condition of t. Note, in particular, that the base point location is only restricted for the return step, since the return value is determined by its state.

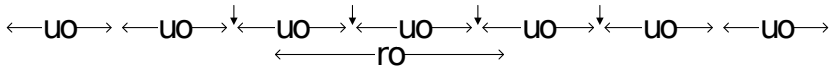


Figure 3: Possible locations of *ro*'s regularity base points.

Function *writeSafe(val)*	**Function** *writeUnsafe(val)*
$i \leftarrow$ ***read***$(lastPos)$	$i \leftarrow$ ***read***$(lastPos)$
write$(v[i+1], val)$	***write***$(lastPos, i+1)$
write$(lastPos, i+1)$	***write***$(v[i+1], val)$

Algorithm 2: Unlike *writeUnsafe*, *writeSafe* ensures a regularity base point for every local state of *readLast*; it guarantees that any concurrent *readLast* operation sees values of $lastPos$ and $v[tmp]$ that occur in the same sequentially reachable post-state. It also has a single visible mutation point (as defined in Section 5), and hence linearizability is established.

In Algorithm 2 we see two versions of an update operation: *writeSafe* guarantees the existence of a base point for every local state of *readLast* (Algorithm 1), and *writeUnsafe* does not. As shown in the full paper, *writeUnsafe* can cause a concurrent *readLast* operation interleaved between its two write steps to see values of $lastPos$ and $v[lastPos]$ that do not satisfy *readLast*'s return step's base condition, and to return an uninitialized value.

3.2 Deriving Correctness from Base Points

In the full paper we prove the following theorems:

Theorem 1 (Validity). *If a data structure ds satisfies base point consistency, then ds is valid.*

Theorem 2 (Regularity). *If a data structure ds satisfies regularity base point consistency, then ds is regular.*

4 Using Our Methodology

We now demonstrate the simplicity of using our methodology. Based on Theorems 1 and 2 above, the proof for correctness of a data structure (such as a

linked list) becomes almost trivial. We look at three linked list implementations: Algorithm 3, which assumes managed memory (i.e., automatic garbage collection), an algorithm that uses RCU methodology, and an algorithm based on hand-over-hand locking (the latter two are deferred to the full paper for space limitations).

For Algorithm 3, we first prove that the listed predicates are indeed base conditions, and next prove that it satisfies regularity base point consistency. By doing so, and based on Theorem 2, we get that the algorithm satisfies both validity and regularity.

```
Function remove(n)
    p ← ⊥
    next ← read(head.next)
    while next ≠ n
        p ← next
        next ← read(p.next)
    write(p.next, n.next)

Function insertLast(n)
    last ← readLast()
    write(last.next, n)
```

Base conditions:

Φ_1 : *true*

Φ_2 : $head \overset{*}{\Rightarrow} n$

Φ_3 : $head \overset{*}{\Rightarrow} n$

```
Function readLast()
    n ← ⊥
    next ← read(head.next)
    while next ≠⊥
        n ← next
        next ← read(n.next)
    return(n)
```

Algorithm 3: A linked list implementation in a memory-managed environment. For simplicity, we do not deal with boundary cases: we assume that a node can be found in the list prior to its deletion, and that there is a dummy head node.

Consider a linked list node stored in local variable n (we assume the entire node is stored in n, including the value and *next* pointer). Here, $head \overset{*}{\Rightarrow} n$ denotes that there is a set of shared variables $\{head, n_1, ..., n_k\}$ such that $head.next = n_1 \wedge n_1.next = n_2 \wedge ... \wedge n_k = n$, i.e., that there exists some path from the shared variable *head* to n. Note that n is the only element in this predicate that is associated with a specific read value. We next prove that this defines base conditions for Algorithm 3.

Lemma 3. *In Algorithm 3, Φ_i defined therein is a base condition of the i-th step of* readLast.

Proof. For Φ_1 the claim is vacuously true. For Φ_2, let l be a local state where *readLast* is about to perform the second read step in *readLast*'s code, meaning that $l(next) \neq \perp$. Note that in this local state both local variables n and *next* hold the same value. Let s be a shared state in which $head \overset{*}{\Rightarrow} l(n)$. Every sequential execution from s iterates over the list until it reaches $l(n)$, hence the same local state where $n = l(n)$ and $next = l(n)$ is reached.

For Φ_3, Let l be a local state where *readLast* has exited the while loop, hence $l(n).next = \perp$. Let s be a shared state such that $head \overset{*}{\Rightarrow} l(n)$. Since $l(n)$ is

reachable from *head* and $l(n).next = \perp$, every sequential execution starting from s exits the while loop and reaches a local state where $n = l(n)$ and $next = \perp$. □

Lemma 4. *In Algorithm 3, if a node n is read during concurrent execution μ of* readLast*, then there is a shared state s in μ such that n is reachable from* head *in s and readLast is pending.*

Proof. If n is read in operation *readLast* from a shared state s, then s exists concurrently with *readLast*. The operation *readLast* starts by reading *head*, and it reaches n.

Thus, n must be linked to some node n' at some point during *readLast*. If n was connected (or added) to the list while n' was still reachable from the head, then there exists a state where n is reachable from the head and we are done. Otherwise, assume n is added as the next node of n' at some point after n' is already detached from the list. Nodes are only added via *insertLast*, which is not executed concurrently with any *remove* operation. This means nodes cannot be added to detached elements of the list. A contradiction. □

The following lemma, combined with Theorem 2 above, guarantees that Algorithm 3 satisfies regularity.

Lemma 5. *Every local state of* readLast *in Algorithm 3 has a regularity base point.*

Proof. We show regularity base points for predicates Φ_i, proven to be base points in Lemma 3.

The claim is vacuously true for Φ_1. We now prove for Φ_2 and Φ_3 : $head \stackrel{*}{\Rightarrow} n$. By Lemma 4 we get that there is a shared state s where $head \stackrel{*}{\Rightarrow} n$ and *readLast* is pending. Note that n's next field is included in s as part of n's value. Since both update operations - *remove* and *insertLast* - have a single write step, every shared state is a post-state of an update operation. Specifically this means that s is a sequentially reachable post-state, and because *readLast* is pending, s is one of the possible regularity base points of *readLast*. □

5 Linearizability

We first show that regularity base point consistency is insufficient for linearizability. In Figure 4 we show an example of a concurrent execution where two read-only operations ro_1 and ro_2 are executed sequentially, and both have regularity base points. The first operation, ro_1, reads the shared variable *first name* and returns Joe, and ro_2 reads the shared variable *surname* and returns Doe. An update operation *uo* updates the data structure concurrently, using two write steps. The return step of ro_1 is based on the post-state of *uo*, whereas ro_2's return step is based on the pre-state of *uo*. There is no sequential execution of the operations where ro_1 returns Joe and ro_2 returns Doe.

Thus, an additional condition is required for linearizability. We suggest *single visible mutation point (SVMP)*, which adds a restriction regarding the behaviour

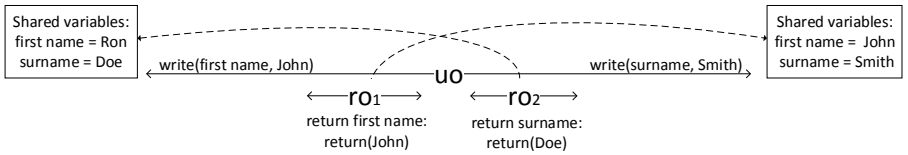


Figure 4: Every local state of ro_1 and ro_2 has a regularity base point, and still the execution is not linearizable. If ro_1 and ro_2 belong to the same process, then the execution is not even sequentially consistent (see the full paper).

of update operations. A data structure that satisfies SVMP and regularity base point consistency is linearizable.

The SVMP condition is related to the number of *visible mutation points* an execution of an update operation has. Intuitively, a visible mutation point in an execution of an update operation is a write step that writes to a shared variable that might be read by a concurrent operation. A more formal definition ensues.

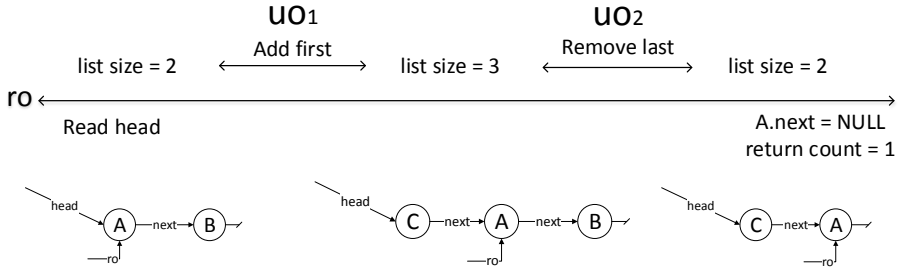
Let α be an execution fragment of op starting from shared state s. We define α^t as the shortest prefix of α including t steps of op, and we denote by $steps_{op}(\alpha)$ the subsequence of α consisting of the steps of op in α. We say that α^t and α^{t-1} are *indistinguishable* to a concurrent read-only operation ro if for every concurrent execution μ_t starting from s and consisting only of steps of ro and α^t, and concurrent execution μ_{t-1} starting from s and consisting only of steps of ro and α^{t-1}, $steps_{ro}(\mu_t) = steps_{ro}(\mu_{t-1})$. In other words, ro's executions are not unaffected by the t'th step of op.

If α^t and α^{t-1} are indistinguishable to a concurrent read-only operation ro, then point t is a *silent point* for ro in α. A point that is not silent is a *visible mutation point* for ro in α.

Definition 4 (SVMP condition). A data structure ds satisfies the SVMP condition if for each update operation uo of ds, in every execution of uo from every sequentially reachable shared state, uo has at most one visible mutation point, for all possible concurrent read-only operations ro of ds.

Note that a read-only operation may see mutation points of multiple updates. Hence, if a data structure satisfies the SVMP condition and not base point consistency, it is not necessarily linearizable. For example, in Figure 5 we see two sequential single visible mutation point operations, and a concurrent read-only operation ro that counts the number of elements in a list. Since ro only sees one element of the list, it returns 1, even though there is no shared state in which the list is of size 1. Thus, the execution is not linearizable or even regular.

Intuitively, if a data structure ds satisfies the SVMP condition, then all of its shared states are sequentially reachable post-states. If ds also satisfies regularity base point consistency, then the visible mutation point condition guarantees that the order among base points of non-interleaved read-only operations preserves the real time order among those operations.



(a) The initial state. **(b)** uo_1's post-state. **(c)** uo_2's post-state.

Figure 5: Every update operation has a single visible mutation point, but the execution is not linearizable.

In Algorithm 3, the remove operation has a single visible mutation point, which is the step that writes to *p.next*. Thus, from Theorem 6 below, this implementation is linearizable. The theorem is proven in the full paper.

Theorem 6 (Linearizability). *If data structure ds satisfies SVMP and regularity base point consistency, then ds is linearizable.*

6 Conclusions and Future Directions

We introduced a new framework for reasoning about correctness of data structures in concurrent executions, which facilitates the process of verifiable parallelization of legacy code. Our methodology consists of identifying base conditions in sequential code, and ensuring regularity base points for these conditions under concurrency. This yields two essential correctness aspects in concurrent executions – the internal behaviour of the concurrent code, which we call validity, and the external behaviour, in this case regularity, which we have generalized here for data structures. Linearizability is guaranteed if the implementation further satisfies the SVMP condition.

We believe that this paper is only the tip of the iceberg, and that many interesting connections can be made using the observations we have presented. For a start, a natural expansion of our work would be to consider also multi-writer data structures. Another interesting direction to pursue is to use our methodology for proving the correctness of more complex data structures than the linked lists in our examples.

Currently, using our methodology involves manually identifying base conditions. It would be interesting to create tools for suggesting a base condition for each local state. One possible approach is to use a dynamic tool that identifies likely program invariants, as in [10], and suggests them as base conditions. Alternatively, a static analysis tool can suggest base conditions, for example by iteratively accumulating read shared variables and omitting ones that are no

longer used by the following code (i.e., shared variables whose values are no longer reflected in the local state).

Another interesting direction for future work might be to define a synchronization mechanism that uses the base conditions in a way that is both general purpose and fine-grained. A mechanism of this type will use default conservative base conditions, such as verifying consistency of the entire read-set for every local state, or two-phase locking of accessed shared variables. In addition, the mechanism will allow users to manually define or suggest finer-grained base conditions. This can be used to improve performance and concurrency, by validating the specified base condition instead of the entire read-set, or by releasing locks when the base condition no longer refers to the value read from them.

From a broader perspective, we showed how correctness can be derived from identifying inner relations in a sequential code, (in our case, base conditions), and maintaining those relations in concurrent executions (via base points). It may be possible to use similar observations in other models and contexts, for example, looking at inner relations in synchronous protocol, in order to derive conditions that ensure their correctness in asynchronous executions.

And last but not least, the definitions of internal behaviour correctness can be extended to include a weaker conditions than validity, (which is quiet conservative). These weaker conditions will handle local states in concurrent executions that are un-reachable via sequential executions but still satisfy the inner correctness of the code.

Acknowledgements

We thank Naama Kraus, Dahlia Malkhi, Yoram Moses, Dani Shaket, Noam Shalev, and Sasha Spiegelman for helpful comments and suggestions.

References

1. Afek, Y., Matveev, A., Shavit, N.: Pessimistic software lock-elision. In: Proceedings of the 26th International Conference on Distributed Computing. pp. 297–311. DISC'12, Springer-Verlag, Berlin, Heidelberg (2012)
2. Arbel, M., Attiya, H.: Concurrent updates with rcu: Search tree as an example. In: Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing. pp. 196–205. PODC '14, ACM, New York, NY, USA (2014)
3. Bayer, R., Schkolnick, M.: Readings in database systems. chap. Concurrency of Operations on B-trees, pp. 129–139. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1988)
4. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ansi sql isolation levels. SIGMOD Rec. 24(2), 1–10 (May 1995)
5. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1986)
6. Brown, T., Ellen, F., Ruppert, E.: Pragmatic primitives for non-blocking data structures. In: PODC. pp. 13–22 (2013)

7. Chockler, G., Lynch, N., Mitra, S., Tauber, J.: Proving atomicity: An assertional approach. In: Proceedings of the 19th International Conference on Distributed Computing. pp. 152–168. DISC'05, Springer-Verlag, Berlin, Heidelberg (2005)
8. Courtois, P.J., Heymans, F., Parnas, D.L.: Concurrent control with "readers" and "writers". Commun. ACM 14(10), 667–668 (1971)
9. Dice, D., Shalev, O., Shavit, N.: Transactional locking ii. In: Proc. of the 20th International Symposium on Distributed Computing (DISC 2006). pp. 194–208 (2006)
10. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. In: Proceedings of the 21st International Conference on Software Engineering. pp. 213–224. ICSE '99, ACM, New York, NY, USA (1999)
11. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 175–184. PPoPP '08, ACM, New York, NY, USA (2008)
12. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In: Proceedings of the 22Nd ACM Symposium on Parallelism in Algorithms and Architectures. pp. 355–364. SPAA '10, ACM, New York, NY, USA (2010)
13. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. SIGARCH Comput. Archit. News 21(2), 289–300 (May 1993)
14. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (July 1990)
15. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. 28(9), 690–691 (September 1979)
16. Lamport, L.: On interprocess communication. part ii: Algorithms. Distributed Computing 1(2), 86–101 (1986)
17. Lev-Ari, K., Chockler, G., Keidar, I.: On correctness of data structures under reads-write concurrency. Tech. Rep. CCIT 866, EE, Technion (August 2014)
18. McKenney, P.E., Slingwine, J.D.: Read-copy update: using execution history to solve concurrency problems, parallel and distributed computing and systems (1998)
19. Moir, M., Shavit, N.: Concurrent data structures. In: Handbook of Data Structures and Applications, D. Metha and S. Sahni Editors. pp. 47–14 47–30 (2007), chapman and Hall/CRC Press
20. Samadi, B.: B-trees in a system with multiple users. Inf. Process. Lett. 5(4), 107–112 (1976)
21. Scherer, III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing. pp. 240–248. PODC '05, ACM, New York, NY, USA (2005)
22. Shao, C., Welch, J.L., Pierce, E., Lee, H.: Multiwriter consistency conditions for shared memory registers. SIAM J. Comput. 40(1), 28–62 (2011)
23. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing. pp. 204–213. PODC '95, ACM, New York, NY, USA (1995)
24. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 129–136. PPoPP '06, ACM, New York, NY, USA (2006)