IRWIN AND JOAN JACOBS
CENTER FOR COMMUNICATION AND INFORMATION TECHNOLOGIES

On Correctness of Data Structures under Reads-Write Concurrency

Kfir Lev-Ari, Gregory Chockler and Idit Keidar

CCIT Report #866
August 2014

DEPARTMENT OF ELECTRICAL ENGINEERING
TECHNION - ISRAEL INSTITUTE OF TECHNOLOGY, HAIFA 32000, ISRAEL

CCIT Report #866 August 2014

On Correctness of Data Structures under Reads-Write Concurrency *

Kfir Lev-Ari
EE Department
Technion,
Haifa, Israel
kfirla@campus.technion.ac.il

Gregory Chockler
CS Department
Royal Holloway,
University of London, UK
gregory.chockler@rhul.ac.uk

Idit Keidar
EE Department
Technion,
Haifa, Israel
idish@ee.technion.ac.il

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, rather, it can express a range of conditions. Here we focus on two new criteria: *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 and sequential consistency. 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 [14, 25] and coarse-grained locking, which do not take into account the inner workings of a specific data structure, are out-performed by hand-tailored fine-grained solutions [21]. 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 [23]. It is therefore not surprising that many approaches do not allow write-write concurrency; these include the *read-copy-update (RCU)* approach [20], flat-combining [13], coarse-grained readers-writer locking [9], 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 [20] and when writes contend heavily [13]. We focus here on solutions that allow only read-read and read-write concurrency.

*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.

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 [22, 4].

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 [10], through multi-versioned concurrency control [6], 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 avoid returning old values, we restrict the locations of the base points that can potentially have effect on the return value of a read-only operation ro to coincide with the return event of an update operation which either immediately precedes, or is executed concurrently with ro. Somewhat surprisingly, this does not suffice for the commonly-used correctness criterion of *linearizability (atomicity)* [15] or even *sequential consistency* [16]. 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. We proceed to exemplify our methodology for a standard linked list implementation, in Section 4. In Section 5 we turn to extend the result for linearizability. We introduce two alternative criteria: the first one, called *linearizability base point consistency* is a direct generalisation of regularity base point consistency further restricting the base points of non-overlapping read-only operation to respect their

real-time order. The second one requires the update operations to satisfy a novel *single visible mutation point (SVMP)* condition, 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 7.

Comparison with Other Approaches The regularity correctness condition was introduced by Lamport [17] 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 [12] 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 [5] 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 [10], 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 [20], 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 [13] and read-write locking [9], 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 [8], or using rely-guarantee reasoning [26]. 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 [7]. 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 an atomic transition label, which can be *invoke*, $a \leftarrow$ *read*(x_i), *write*(x_i,v), or *return*(v):
    - An *invoke* changes the initial local state $\perp$ into another local state, and does not change the shared state.
    - A $write(x_i, v)$ changes the local state and changes the value of shared variable $x_i \in X$ to v.
    - A $a \leftarrow read(x_i)$ reads the value of one variable $x_i \in X$ from the shared state and changes the local state accordingly (i.e., stores the value of x_i in a local variable a).
    - A $return(v)$ ends the operation by changing the local state to $\perp$ and returning v to the calling process. It does not change the shared state.

    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* from a shared state $s_i \in \mathcal{S}$ is a sequence of transitions of the form:

$$\underset{s_i}{\overset{\perp}{}},\ invoke,\ \underset{s_i}{\overset{l_1}{}},\ step_1,\ \underset{s_{i+1}}{\overset{l_2}{}},\ step_2,\ \ldots,\ \underset{s_j}{\overset{l_k}{}},\ return_k,\ \underset{s_j}{\overset{\perp}{}},$$

where $\tau(l_m,\ s_n) = \langle step_m,\ l_{m+1},\ s_{n+1} \rangle$. The first step is invoke, ensuing steps are read or write steps, and the last step is a return step.

A *sequential execution of a data structure* is a (finite or infinite) sequence μ:

$$\mu = \underset{s_1}{\overset{\perp}{}},\ O_1,\ \underset{s_2}{\overset{\perp}{}},\ O_2,\ \ldots,$$

where $s_1 \in \mathcal{S}_0$ and every $\overset{\bot}{s_j}$, O_j, $\overset{\bot}{s_{j+1}}$ in μ is a sequential execution of some operation. If μ is finite, it can end after an operation or during an operation. In the latter case, we say that the last operation is *pending* in μ. Note that in a sequential execution there can be at most one pending operation.

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.

For example, the following is a concurrent execution fragment that starts from a shared state s_i and invokes two operations: O_A and O_B. The first operation takes a write step, and then O_B takes a read step. We subscript every step and local state with the operation it pertains to.

$$\underset{s_i}{\emptyset},\ invoke_A(),\ \underset{s_i}{\{l_{1,A}\}},\ write_A(x_i, v),\ \underset{s_{i+1}}{\{l_{2,A}\}},\ invoke_B(),\ \underset{s_{i+1}}{\{l_{2,A},\ l_{1,B}\}},\ a \leftarrow read_B(x_i),\ \underset{s_{i+1}}{\{l_{2,A}, l_{2,B}\}}.$$

In the remainder of this paper we assume that for all concurrent executions μ of the date structure, and any two update operations uo_1 and uo_2 invoked in μ, uo_1 and uo_2 are not executed concurrently to each other (i.e., either uo_1 is invoked after uo_2 returns, or vice versa).

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 [12] 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 A history H_σ is *linearizable* [15] 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.

Regularity We next extend Lamport's regular register definition [17] for data structures (we do not discuss regularity for executions with concurrent update operations, which can be defined similarly to [24]). 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.

Sequential Consistency A history H_σ is *sequentially consistent* [16] 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 belong to the same process, appear in the same order in σ and in π. A data structure *ds* is *sequentially consistent*, if for every execution σ of *ds*, H_σ is sequentially consistent.

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 our notion of validity may still be correct in a weaker sense, e.g., if allowed to abort an operation, which encountered a sequentially unreachable state. We do not address such an alternative notions of correctness in our discussion.

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 the 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.

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

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

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. The base conditions for every possible local state of *readLast* are detailed in Algorithm 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.

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

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 3 for corresponding update operations.

Shared variables: $lastPos,\ \forall i \in \mathbb{N} : v[i]$

base condition	***step***
$\Phi_1 : true$	$tmp \leftarrow \textbf{\textit{read}}(lastPos)$
$\Phi_2 : lastPos = tmp$	$res \leftarrow \textbf{\textit{read}}(v[tmp])$
$\Phi_3 : lastPos = tmp \wedge v[tmp] = res$	$\textbf{\textit{return}}(res)$

Algorithm 2: ReadLast operation. The shared variable *lastPos* is the index of the last updated value in array v. See Algorithm 3 for the corresponding update operation.

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.

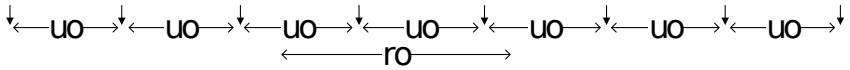


Figure 2: Possible locations of *ro*'s base points.

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: (1) an update operation that returns in μ after ro's invoke step and before ro's return step; or (2) the last update operation that returns 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 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 that the base point location is only restricted for the return step, since the return value is determined by its state.

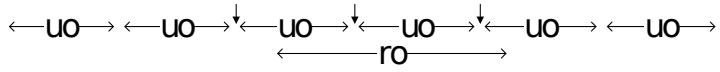


Figure 3: Possible locations of *ro*'s regularity base points.

In Algorithm 3 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 Section 3.2, *writeUnsafe* can cause a concurrent *readLast* operation interleaved between its two write steps to see

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

Algorithm 3: 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.

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 Satisfying the Regularity Base Point Consistency

Let us examine the possible concurrent executions an invocation *ro* of *readLast* (Algorithm 1) and an invocation *uo* of *writeSafe* (Algorithm 3) with parameter 80 starting from s_1 (Figure 1). There are four possible interleavings of write steps of *uo* and read steps of *ro* starting from s_1 shown in Algorithm 4. In each of them, *ro* returns 7, and s_1 is the base point of its last local state.

μ_1 :	μ_2 :	μ_3 :	μ_4 :
$\boldsymbol{read_{ro}}(lastPos)$	$\boldsymbol{read_{ro}}(lastPos)$	$read_{uo}(lastPos)$	$read_{uo}(lastPos)$
$read_{uo}(lastPos)$	$read_{uo}(lastPos)$	$write_{uo}(v[2], 80)$	$write_{uo}(v[2], 80)$
$write_{uo}(v[2], 80)$	$write_{uo}(v[2], 80)$	$\boldsymbol{read_{ro}}(lastPos)$	$\boldsymbol{read_{ro}}(lastPos)$
$\boldsymbol{write_{uo}}(lastPos, 2)$	$read_{ro}(v[1])$	$\boldsymbol{write_{uo}}(lastPos, 2)$	$read_{ro}(v[1])$
$read_{ro}(v[1])$	$return_{ro}(7)$	$read_{ro}(v[1])$	$return_{ro}(7)$
$return_{ro}(7)$	$\boldsymbol{write_{uo}}(lastPos, 2)$	$return_{ro}(7)$	$\boldsymbol{write_{uo}}(lastPos, 2)$

Algorithm 4: Four interleaved executions of invocation *ro* of *readLast* and invocation *uo* of *writeSafe* that start from s_1.

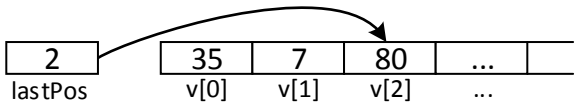


Figure 4: The shared state s_1'. It is the post-state after executing *writeSafe* or *writeUnsafe* from s_1 (Figure 1) with initial value 80.

Now let us examine a concurrent execution consisting of *readLast* and *writeUnsafe* (Algorithm 3), in which *readLast* reads a value from *lastPos* right after *writeUnsafe* writes to it. In Algorithm 5 we see such an execution that starts from s_1. The last local state of *ro* is $l_3' = \{tmp = 2, res = 99\}$. Neither s_1 and s_1' (Figure 4) satisfies $\Phi_3' : lastPos = 2 \wedge v[2] = 99$, meaning that l_3' does not have a base point with Φ_3'.

$$
\begin{array}{l}
read_{seq}(lastPos) \\
\boldsymbol{write_{seq}}(lastPos, 2) \\
\boldsymbol{read_{ro}}(lastPos) \\
read_{ro}(v[2]) \\
return_{ro}(99) \\
write_{seq}(v[2], 80)
\end{array}
$$

Algorithm 5: A possible concurrent execution consisting of *readLast* and *writeUnsafe*, starting from s_1.

Below we show that this is not an artifact of our choice of a base condition – we prove that for every base condition Φ_3' of l_3', both $\Phi_3'(s_1)$ and $\Phi_3'(s_1')$ are false.

Lemma 1. *A data structure that has both writeUnsafe and readLast is not regularity base point consistent.*

Proof. Given the execution of Algorithm 5 that starts from the shared state s_1 and ends in shared state s_1', we assume by contradiction that there is such a base condition of $l_3' = \{tmp = 2, res = 99\}$ that is satisfied by s_1 or s_1'. By the definition of base condition, if we execute *readLast* sequentially from a shared state that satisfies l_3''s base condition, we reach l_3'. But if we execute *readLast* from s_1 we reach $l_3 : \{tmp = 1, res = 7\}$ and if we execute from s_1' we reach $l_3'' : \{tmp = 2, res = 80\}$. A contradiction. □

3.3 Deriving Correctness from Base Points

We start by proving that the base point consistency implies validity.

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

Proof. In order to prove that *ds* is valid, we need to prove that for every execution μ of *ds*, for any operation $op \in \mu$ of *ds*, every local state is sequentially reachable. If μ is a sequential execution then the claim holds. If *op* is an update operation, since every update operation is executed sequentially starting from a sequentially reachable post-state, then every local state of *op* is sequentially reachable. Now we prove for *op* that is a read-only operation in concurrent execution μ. Given that the data structure satisfies the base point consistency, every local state l of every read-only operation in μ has a base point s_{base}. In order to show that l is sequentially reachable, we build a sequential execution μ' that starts from the same initial state as μ and consists of the same update operations that appear in μ until s_{base}. Then we add a sequential execution of *op*. Since s_{base} is a base point of l, l is reached in μ' and therefore is sequentially reachable. □

We now prove that the regularity base point consistency implies regularity.

Lemma 2. *Let μ be a concurrent execution of a data structure* ds*. Let ro be a read-only operation of* ds *executed in μ, which returns v. If* ds *satisfies regularity base point consistency then there exists a sequentially reachable shared state s in μ such that: (1) s is the post-state of some update operation that is either concurrent with ro or is the last before ro is invoked; and (2) when executing ro from s, its return value is equal to v.*

Proof. Let l be the local state that precedes ro's return step. Since τ is deterministic, its return value v is fully determined by l, and every execution of ro that reaches l returns v. Given that *ds* satisfies regularity base point consistency, l, which is the last local state of ro, has a regularity base point for some base condition Φ of l. Let s denote a base point of l for Φ in μ. By the definition of a regularity base point, the shared state s is the post-state of some update operation that is either concurrent with ro or is the last before ro is invoked, and $\Phi(s)$ is true. By the definition of base condition Φ, we get that l is reached in ro's sequential execution from s, that is, when ro is sequentially executed from s, its return value is v. □

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

Proof. In order to prove that *ds* is regular, we need to show that for every concurrent execution μ of *ds* with history H_μ, for any read-only operation $ro \in H_\mu$, if we omit all other read-only operations from H_μ, the resulting history H_μ^{ro} is linearizable. Recall that update operations are executed sequentially.

If μ includes only update operations then μ vacuously satisfies the condition. Otherwise, let ro be a read-only operation in μ. If ro is pending in μ, we build a sequential history by removing ro's invocation from H_μ^{ro}, which is allowed by the definition of linearizability.

Consider now a read-only operation ro that returns in μ. Since every return step of ro has a regularity base point in μ, by Lemma 2, we get that there is a shared state s in μ from which ro's sequential execution returns the same value as in μ, and s is the post-state of some update operation that is either concurrent with ro or is the last before ro is invoked. We build a sequential execution μ_{seq}^{ro} from the sequence of update operations in μ with ro added at point s. Then μ_{seq}^{ro} is a sequential execution of *ds*, which belongs to the sequential specification. Every pair of operations that are not interleaved in μ appear in the same order in μ_{seq}^{ro}. Therefore, H_μ^{ro} is linearizable. □

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 – one assuming managed memory, (i.e., automatic garbage collection, Algorithm 6), one using read-copy-update methodology (Algorithm 7), and one using hand-over-hand locking (Algorithm 8).

For Algorithm 6, we first prove that the listed predicates are indeed base conditions, and next we prove that it satisfies the base point consistency and the regularity base point consistency. By doing so, and based on Theorems 1 and 2, we get that the algorithm satisfy both validity and regularity.

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, $head \stackrel{*}{\Rightarrow} n$ 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 6.

Lemma 3. *In Algorithm 6, 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 $head \stackrel{*}{\Rightarrow} l(n)$. 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 $head \stackrel{*}{\Rightarrow} l(n)$. 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 6, 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. □

```
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:

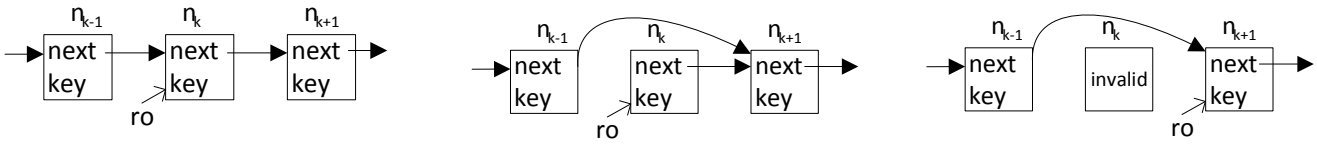
$\Phi_1 : true$

$\Phi_2 : head \stackrel{*}{\Rightarrow} n$

$\Phi_3 : head \stackrel{*}{\Rightarrow} n$

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

Algorithm 6: 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.



(a) *rcuRemove(n_k)*'s pre-state. **(b)** The shared state during n_k's removal. **(c)** *rcuRemove(n_k)*'s post-state.

Figure 5: Shared states in a concurrent execution consisting of *rcuRemove(n_k)* and *rcuReadLast* (*ro*).

The following lemma, combined with Theorem 2 above, guarantees that Algorithm 6 satisfies regularity.

Lemma 5. *Every local state of* readLast *in Algorithm 6 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*. □

RCU Read-copy-update [20] is a synchronization strategy that aims to reduce read operations' synchronization overhead as much as possible, while risking a high synchronization overhead for update operations. The idea is that only update operations require locks, and the writes mutate the data structure in a way that ensures that concurrent readers always see a consistent view. Additionally, writers do not free data while it is used by readers. Note that RCU does not allow write-write concurrency.

RCU is commonly used via primitives that resemble readers-writer locks [3]: *rcuReadLock* and *rcuReadUnLock*. There are other primitives that encapsulate list traversal, but we do not use them in our example since we wish to illustrate the general approach. Instead, we use primitives that are commonly used for creating RCU-protected non-list data structures (such as arrays and trees): *rcuWrite(p, v)* (originally called *rcuAssignPointer*), and *rcuRead(p)* (originally called *rcuDereference*) [19].

```
Function rcuRemove(n)
    p ← ⊥
    next ← read(head.next)
    while next ≠ n
        p ← next
        next ← read(p.next)
    rcuWrite(p.next, n.next)
    rcuWaitForReaders()
    write(n, invalid)

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

```
Base conditions:    Function rcuReadLast()
                        rcuReadLock()
                        n ←⊥
Φ1 : true               next ← rcuRead(head.next)
                        while next ≠⊥
                            n ← next
Φ2 : head ⇒* n              next ← rcuRead(n.next)
                        rcuReadUnlock()
Φ3 : head ⇒* n          return(n)
```

Algorithm 7: An RCU linked list implementation. 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.

In Algorithm 7, *rcuWrite* is a write step that changes the next pointer of n's predecessor, and it occurs between the shared states (a) and (b) in Figure 5. The invalidation of n takes place once all read-only operations that use n no longer hold a reference to it, as guaranteed by *rcuWaitForReaders*(). The latter

happens between the shared states of (b) and (c). The *rcuReadLast* operation holds at most a single reference to list node at a given time, and our base condition links *head* to it. We see in Figure 5 that invalid nodes are unreachable from *head* in sequentially reachable post-states. Thus, the base condition $head \stackrel{*}{\Rightarrow} n$ implies that *ro* never holds a pointer to an invalid node.

The correctness of the base conditions annotated in Algorithm 7 follows the same reasoning as Lemma 3, and hence we omit it here. We now prove that Algorithm 7 satisfies regularity base point consistency, and therefore by Theorems 1 and 2, Algorithm 7 satisfies validity and regularity.

Lemma 6. *In Algorithm 7, if a node n is read during concurrent execution μ of* rcuReadLast*, then there is a state where the shared state is s in μ such that n is reachable from* head *in s and ro is pending.*

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

Thus, n must be linked to some node n' at some point during *rcuReadLast*. 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 *rcuRemove* operation. This means nodes cannot be added to detached elements of the list. A contradiction. □

Lemma 7. *Every local state of* rcuReadLast *in Algorithm 7 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 : $head \stackrel{*}{\Rightarrow} n$. Every read step is encapsulated by *rcuRead*, and is surrounded by *rcuReadLock* and *rcuReadUnlock*. These calls guarantee that as long as the reader holds a reference to the value it read using *rcuRead*, the value cannot be changed by the write step of *rcuRemove* that removes a node from the list. In addition, *rcuRemove* waits for all readers to forget a node before invalidating it, and invalidates it only after the node is not reachable. Therefore, it is guaranteed that every node that is read is valid. In addition, Lemma 6 guarantees that there is a shared state s where $head \stackrel{*}{\Rightarrow} n$ and *rcuReadLast* is pending. Note that n's next field is included in s as part of n's value. Since the invalidation is not visible to the readers, the post-state of *rcuRemove* and the shared state after *rcuWaitForReaders()* are indistinguishable to the readers. The operation *insertLast* has one write step have a single write step and therefore it is always found between two sequentially reachable shared states.

In conclusion, every shared state is a post-state of an update operation from every reader perspective. Specifically this means that s is a sequentially reachable post-state, and because *rcuReadLast* is pending, s is one of the possible regularity base points of *rcuReadLast*. □

hand-over-hand locking In *hand-over-hand locking*, a data structure is traversed by holding a lock to the next node in the traversal before unlocking the previous one.

In Algorithm 8 we give a linked list implementation using hand-over-hand locking. The locks used therein are readers-writer locks [18], where write locks are exclusive and multiple threads can obtain read locks concurrently. We define a lock for every shared variable $x_i \in X$, and extend the model with $lock(x_i)$ and $unlock(\{x_{i_1}, x_{i_2}, ...\})$ steps. The correctness of the base conditions annotated in Algorithm 8 follows the same reasoning as Lemma 3, and hence we omit it here. The reachable post-states in Figure 5 are (a) and (c). State (b) does not occur in this implementation since *ro* cannot access n concurrently with an update operation that holds n's lock. In the following lemma we prove that Algorithm 8 satisfies regularity base point consistency.

Lemma 8. *In Algorithm 8, if a node n is read during concurrent execution μ of* hohReadLast*, then there is a state where the shared state is s in μ such that n is reachable from* head *in s and ro is pending.*

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

```
Function hohRemove(n)
    p ← ⊥
    lock(head.next)
    next ← read(head.next)
    while next ≠ n
        p ← next
        lock(p.next)
        unlock(p)
        next ← read(p.next)
    write(p.next, n.next)
    lock(n)
    invalidate(n)
    unlock(n, p)

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

Base conditions:

$\Phi_1 : true$

$\Phi_2 : head \stackrel{*}{\Rightarrow} n$

$\Phi_3 : head \stackrel{*}{\Rightarrow} n$

```
Function hohReadLast()
    n ← ⊥
    lock(head.next)
    next ← read(head.next)
    while next ≠⊥
        n ← next
        lock(n.next)
        next ← read(n.next)
        unlock(n)
    unlock(next)
    return(n)
```

Algorithm 8: A linked list implementation using hand-over-hand locking. 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.

Thus, n must be linked to some node n' at some point during *hohReadLast*. 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 *hohRemove* operation. This means nodes cannot be added to detached elements of the list. A contradiction. □

Lemma 9. *Every local state of* hohReadLast *in Algorithm 8 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 $\Phi_3 : head \stackrel{*}{\Rightarrow} n$. In *hohReadLast*, the reader reads a node only after locking it. Thus, the invalidation of that node is not visible to the readers due to the locking that *hohRemove* performs before any write step, meaning that the post-state of *hohRemove* and the shared state after the first write step of *hohRemove* are indistinguishable to the readers. Therefore, the reader only sees valid nodes. In addition, Lemma 8 guarantees that there is a shared state s where $head \stackrel{*}{\Rightarrow} n$ and *hohReadLast* is pending. Note that n's next field is included in s as part of n's value.

The operation *insertLast* has one write step have a single write step and therefore it is always found between two sequentially reachable shared states.

In conclusion, every shared state is a post-state of an update operation from every reader perspective. Specifically this means that s is a sequentially reachable post-state, and because *hohReadLast* is pending, s is one of the possible regularity base points of *hohReadLast*. □

5 Linearizability

We first show that regularity base point consistency is insufficient for linearizability. In Figure 6 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.

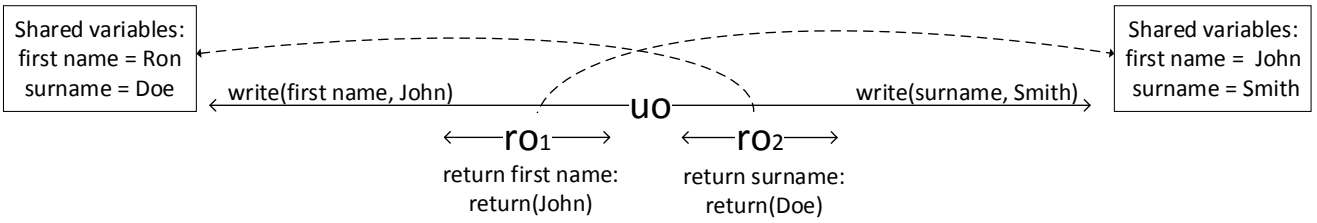


Figure 6: Every local state of ro_1 and ro_2 has a regularity base point, and still the execution is not linearizable. If ro_1 and ro_2 belong to the same process, then the execution is not even sequentially consistent.

Thus, an additional condition is required for linearizability. We suggest two possibilities: the first is *linearizability base point consistency* - this condition adds a restriction to the possible locations of the regularity base points, and is suffice for linearizability by itself. The second is *single visible mutation point (SVMP)*, which adds a restriction regarding the behaviour of update operations. A data structure that satisfies SVMP and regularity base point consistency is linearizable.

5.1 Linearizability Base Point Consistency

Recall that in order to satisfy linearizability, a data structure needs to guarantee that for every concurrent execution μ there is an equivalent sequential execution μ_{seq} such that the order between non-interleaved operations in μ is preserved in μ_{seq}. One way to ensure this is to determine that the order between the regularity base points has to follow the order of non-interleaved read-only operations.

We say that a data structure *ds* satisfies *linearizability base point consistency* if it satisfies the regularity base point consistency, and for every concurrent execution μ in which a read-only operation ro_1 of *ds* precedes a read-only operation ro_2 of *ds*, the return step of ro_1 has a regularity base point in μ that precedes or equals to ro_2's return step's regularity base point in μ.

Notice that base point consistency, regularity base point consistency and linearizability base point consistency, are a sequence in which each condition is a subset of the previous one in terms of possible base point locations. This construction of criteria for data structures correctness follow the construction of Lamport for safe, regular and atomic registers [17]. The connection between regular and linearizable data structures, (as defined by regularity and linearizability base point consistency), reflects the one between regular and atomic registers. Notice that safe data structure can be defined in the same sense.

Theorem 3 (Linearizability)**.** *If a data structure* ds *satisfies linearizability base point consistency, then* ds *is linearizable.*

Proof. Given a concurrent execution μ of *ds*, we create a total ordering σ of operations in μ as follows:

- The order of the update operations in σ is the same as their order in μ.
- Let ro be a read-only operation in μ which returns v. Since *ds* satisfies regularity base point consistency, by Lemma 2, there exists a sequentially reachable state s, which is a post-state of an update operation uo such that uo is either concurrent with ro, or returns before ro is invoked, and the return value of ro is v if it is executed sequentially from s. We therefore, insert immediately the return step of uo, and (2) completes before any uo' which follows uo in σ is invoked. If two read-only operations ro_1 and ro_2 share the same post-state s, and ro_1 returns before ro_2 is invoked in μ, then ro_1 is placed before ro_2 in σ. Otherwise, the order between them is arbitrary provided they are inserted sequentially one after the other, and none of them is inserted after uo' is invoked.

Since by Lemma 2, the return value of each ro is the same as the one that will be returned in a sequential execution of ro if it is invoked after the write operation immediately preceding ro in σ, σ is a valid sequential execution of *ds*.

It remains to show that σ preserves the relative order of each pair of non-overlapping operations in μ. First, it is easy to see that the order of each pair of operations op_1 and op_2 such that either both op_1 and op_2 are updates, or exactly one of them is an update, and the the other one is read-only is the same in both μ and σ.

Let ro_1 and ro_2 be two complete read-only operations in μ such that ro_1 returns before ro_2 is invoked in μ; and ro_1's base point s_1 is distinct from ro_2's base point s_2 such that s_1 and s_2 are post-states of update operations uo_1 and uo_2 respectively.

Assume by way of contradiction that ro_2 precedes ro_1 in σ. By construction of σ, uo_2, (resp., uo_1), is the last update operation preceding ro_2 (resp., ro_1). Also, by construction, uo_2 must precede uo_1 in μ, and their respective post-states s_2 and s_1 are base points of ro_2 and ro_1 respectively. However, since s_2 is reached earlier than s_1, by linearizability base point consistency, ro_2 must precede ro_1 in μ, which is a contradiction.

We conclude that σ is a valid sequential execution of *ds*, which preserves the order of all non-overlapping operations in μ. Therefore, the history H of σ belongs to the sequential specification of *ds*, and preserves the order of all non-overlapping operations in μ. Hence, H is a linearization of μ. □

5.2 SVMP

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.

SVMP combined with regularity base point consistency ensures linearizability. The set of data structures that satisfy those two conditions is a subset of the set of data structures that satisfy linearizability base point consistency, (i.e., if SVMP and regularity base point consistency is satisfied then linearizability base point consistency is satisfied as well, but the opposite is not always true). The reason for defining SVMP is that it has weaker demands from the read operations and it is easier to work with.

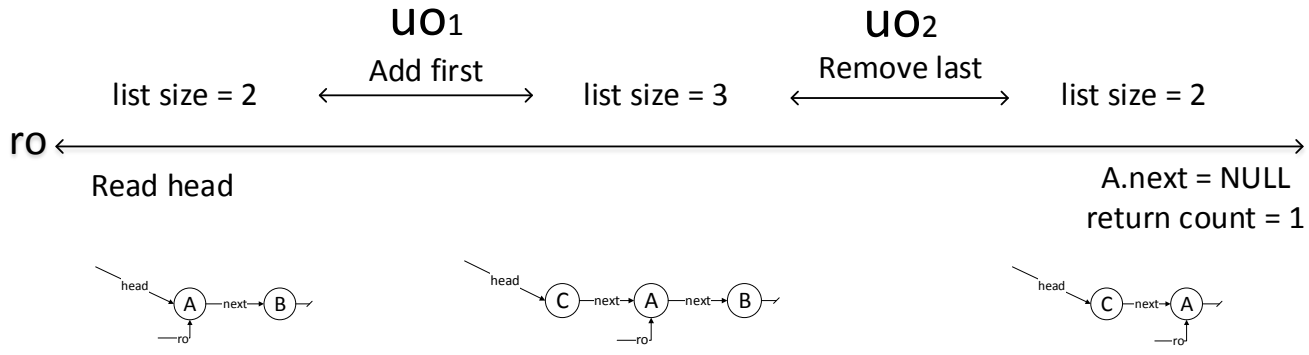
Let α be an execution fragment of *op* starting from a shared state s. We define α^t as the shortest prefix of α including t steps of *op*, and we denote by $steps_{op}(\alpha)$ the subsequence of α consisting of the steps of *op* in α. We say that α^t and α^{t-1} are *indistinguishable* to a concurrent 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 affected 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: (1) uo has at most one visible mutation point, for all possible concurrent read-only operations ro of* ds*; and (2) if two points MP and MP' of uo are the visible mutation points for ro and ro′ respectively, then MP=MP'.*

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 7 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 each SVMP's post-state and the operation's post-state are indistinguishable from the perspective of any concurrent read-only operation. If *ds* also satisfies regularity base point consistency, then the visible mutation point condition guarantees that



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

Figure 7: Every update operation has a single visible mutation point, but the execution is not linearizable.

the order among base points of non-interleaved read-only operations preserves the real time order among those operations.

In Algorithm 6, the remove operation has a single visible mutation point, which is the step that writes to *p.next*. Thus, from Theorem 4 below, this implementation is linearizable.

Lemma 10. *If a data structure* ds *satisfies the SVMP condition, then for every execution* μ *of* ds*, for every update operation* uo *in* μ*, the post-state of* uo *satisfies a base condition of a step* s *of operation* o *if and only if the post-state of* uo*'s SVMP satisfies a base condition of* s*.*

Proof. By SVMP definition, each step in $\text{steps}_{uo}(\mu)$ after the SVMP of uo is a silent point. Thus, if we prove that the shared state that is a pre-state of a silent point satisfies a base condition of a step s if and only if its post-state satisfies a base-condition of s as well, the claim will follow.

Assume by way of contradiction, W.L.O.G., that the shared state that is a pre-state of a silent point sp satisfies a base condition of a step s of an operation o and that the post-state of the sp does not satisfy a base condition of s. Thus, by definition of base condition, we can build a concurrent execution σ from some initial shared state that consists of uo's execution until the pre-state of sp is reached and then execution of o until it reaches s.

We can build another concurrent execution that starts from the same shared state as σ, in which uo is executed until it reaches sp, and from sp's post state we execute o. Since sp's post state does not satisfies any base condition of s, the execution of o does not reach s. These two executions are not indistinguishable, in contradiction to sp being a silent point. □

For steps or states n and m, the notation $n <_\mu m$ denotes that n appears before m in μ, and $n \leq_\mu m$ if $n <_\mu m$ or n and m are the same state/step. For operations o_1 and o_2, the notation $o_1 <_\mu o_2$ denotes that o_1 returns before o_2 is invoked in μ.

Theorem 4 (SVMP Linearizability)**.** *If a data structure* ds *satisfies the SVMP condition and the regularity base point consistency then* ds *is linearizable.*

Proof. Let μ be a concurrent execution of *ds*. We build a linearization μ_{seq} of μ in the following way: we start with he sequence of update operations appearing in the same order as in μ, then we add every read-only operation ro from μ as follows: (1) ro is invoked in μ_{seq} from its first regularity base point; and (2) the order of read-only operations that have the same base point follows their invocation order in μ.

Since μ_{seq} is defined as a sequential execution starting from an initial shared state of the *ds*, μ_{seq} satisfies the sequential specification of *ds*.

It is given that every read-only operation in μ has a regularity base point and is executed sequentially from it in μ_{seq}, so every read-only operation's return step is identical in both executions.

Since each read-only operation ro is executed from a regularity base point, ro's order is preserved in μ_{seq} relative to every update operation that dose not overlap ro in μ.

It remains to show that our ordering of read-only operations according to their first regularity base point does not contradict their order in μ. Assume by way of contradiction that there are two read-only operations, ro_1 and ro_2, s.t. $ro_1 <_\mu ro_2$ and $ro_2 <_{\mu_{seq}} ro_1$. Let uo_1 be the operation that has the last SVMP that precedes ro_1's return step in μ, and let uo_2 be the operation with the last SVMP that precedes ro_2's invoke step in μ.

Since $ro_2 <_{\mu_{seq}} ro_1$, by construction of μ_{seq}, $uo_2 <_{\mu_{seq}} uo_1$, and therefore, $\text{SVMP}_2 <_\mu \text{SVMP}_1$. Given that $ro_1 <_\mu ro_2$, there are only two ways they can be positioned relative to SVMP_1 in μ: First, if ro_1 overlaps SVMP_1 and since $\text{SVMP}_2 <_\mu \text{SVMP}_1$, then SVMP_2 is not the latest SVMP preceding ro_2 in μ, and therefore, by Definition 3 it is not a regularity base point. Otherwise, SVMP_1 is necessarily located after the return step of ro_1, and therefore, by Definition 3 SVMP_1 cannot be ro_1's regularity base point. A contradiction. □

6 Sequential Consistency

Some systems use the correctness criterion of sequential consistency [16], which relaxes linearizability by not requiring *real time order* (RTO) between operations of different processes.

Note that sequential consistency and regularity are incomparable: Regularity does not impose RTO on read-only operations even if they belong to the same process, while in sequential consistency, the RTO of read-only operations of the same process is preserved. On the other hand, regularity enforces the RTO between an update operation and every other operation, while sequential consistency allows re-ordering of operations executed by different processes.

We say that a data structure *ds* satisfies *sequentially base point consistency* if it satisfies the base point consistency, and for every concurrent execution μ in which a read-only operation ro_1 of *ds* precedes a read-only operation ro_2 of *ds* and both belong to the same process, the return step of ro_1 has a base point in μ that precedes or equals to ro_2's return step's base point in μ.

We now prove that the loose snapshot condition along with the SVMP condition ensures sequential consistency.

Lemma 11. *Let μ be a concurrent execution such that: (1) μ starts from a sequentially reachable post-state s; and (2) every return step of every read-only operation in μ has a base point ; and (3) for every read-only operation ro_1 that precedes a read-only operation ro_2 of the same process, the return step of ro_1 has a base point in μ that precedes or equals to ro_2's return step's base point in μ.*

Then there is a sequential execution μ_{seq} such that: (1) μ_{seq} and μ contain the same operations; and (2) for every process, all its operations appear in the same order in μ_{seq} and in μ.

Proof. We build a sequential execution μ_{seq} in the following way: (1) μ_{seq} starts from the same shared state s as μ. I t is given that s is sequentially reachable. (2) All update operations in μ appear in the same order in μ_{seq}. (3) Every read-only operation ro in μ is executed in μ_{seq} from a post-state that is a base point of the return step of ro. It is given that for operations of the same process, different base points appear in the execution in the same order as the operations do. Therefore if there are multiple possibilities for a base point, the operation is executed from the base point according to the that order. Read-only operations of the same process that have the same base point are executed from it at the same order in μ_{seq} as in μ. (4) The order of read-only operations that do not belong to the same process and are executed from the same base point is arbitrary.

Since only update operations can change the shared state and their sequential order is the same in both executions, every update operation is executed in μ_{seq} from the same shared state as in μ. By the definitions of base point and base condition we get that every read-only operation in μ_{seq} returns the same value in μ_{seq} as in $\mu - ro$ is executed from a shared state that is a base point of its return step, and the last local state determines ro's return value. □

Theorem 5 (Sequential consistency)**.** *If a data structure* ds *satisfies sequentially base point consistency, then* ds *is sequentially consistent.*

Proof. Let μ be a concurrent execution of *ds*. By Lemma 11 we get that there is a sequential execution μ_{seq}, such that $H_{\mu_{seq}}$ is a permutation of complete(H_μ) that belongs to the sequential specification of *ds* and keeps the RTO of operations that belong to the same process in μ. Thus *ds* is sequentially consistent. □

7 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 either the SVMP condition, or linearizability base point consistency.

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 [11], 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] Y. Afek, A. Matveev, and N. Shavit. Pessimistic software lock-elision. In *Proceedings of the 26th International Conference on Distributed Computing*, DISC'12, pages 297–311, Berlin, Heidelberg, 2012. Springer-Verlag.

[2] M. Arbel and H. Attiya. Concurrent updates with rcu: Search tree as an example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 196–205, New York, NY, USA, 2014. ACM.

[3] A. Arcangeli, M. Cao, P. E. McKenney, and D. Sarma. Using read-copy-update techniques for system v ipc in the linux 2.5 kernel. In *USENIX Annual Technical Conference, FREENIX Track*, pages 297–309. USENIX, 2003.

[4] R. Bayer and M. Schkolnick. Readings in database systems. chapter Concurrency of Operations on B-trees, pages 129–139. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[5] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 24(2):1–10, May 1995.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[7] T. Brown, F. Ellen, and E. Ruppert. Pragmatic primitives for non-blocking data structures. In *PODC*, pages 13–22, 2013.

[8] G. Chockler, N. Lynch, S. Mitra, and J. Tauber. Proving atomicity: An assertional approach. In *Proceedings of the 19th International Conference on Distributed Computing*, DISC'05, pages 152–168, Berlin, Heidelberg, 2005. Springer-Verlag.

[9] P.-J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, 1971.

[10] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

[11] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[12] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM.

[13] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22Nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM.

[14] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

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

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

[17] L. Lamport. On interprocess communication. part ii: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[18] P. E. McKenney. Selecting locking primitives for parallel programming. *Commun. ACM*, 39(10):75–82, Oct. 1996.

[19] P. E. McKenney. RCU part 3: the RCU API. January 2008.

[20] P. E. McKenney and J. D. Slingwine. Read-copy update: using execution history to solve concurrency problems, parallel and distributed computing and systems, 1998.

[21] M. Moir and N. Shavit. Concurrent data structures. In *Handbook of Data Structures and Applications, D. Metha and S. Sahni Editors*, pages 47–14 47–30, 2007. Chapman and Hall/CRC Press.

[22] B. Samadi. B-trees in a system with multiple users. *Inf. Process. Lett.*, 5(4):107–112, 1976.

[23] W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 240–248, New York, NY, USA, 2005. ACM.

[24] C. Shao, J. L. Welch, E. Pierce, and H. Lee. Multiwriter consistency conditions for shared memory registers. *SIAM J. Comput.*, 40(1):28–62, 2011.

[25] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[26] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 129–136, New York, NY, USA, 2006. ACM.