Proving Correctness of Concurrent Objects by Validating Linearization Points *

Nandini Singhal, Muktikanta Sa, Ajay Singh, Archit Somani, Sathya Peri
Department of Computer Science & Engineering
{cs15mtech01004, cs15resch11012, cs15mtech01001, cs15resch01001, sathya_p}@iith.ac.in

Abstract

Concurrent data structures (CDS) and algorithms can accelerate the big data applications. But, developing a CDS and verifying its correctness is an uphill task. Thus, in this paper we address the most basic problem of this domain: given the set of LPs of a CDS, how to show its correctness? We assume that we are given a CDS and its LPs(linearisation points). We have developed a hand-crafted technique of proving correctness of the CDSs by validating its LPs. We also believe that this technique might also offer the programmer some insight to develop more efficient variants of the CDS, by iteratively applying the steps of the proposed technique uncovering hidden parallelism in each iteration or errors in CDS design, if any. The proof technique can be applied to prove the correctness of several commonly used CDSs developed in literature such as Lock-free list based Sets, Skiplists etc. To show the efficacy of this technique, we show the correctness of lazy-list and hoh-locking-list based set.

1 Introduction

Concurrent data structures or *CDS* such as concurrent stacks, queues, lists etc. have become very popular in the past few years due to the rise of multi-core systems and due to their performance benefits over their sequential counterparts. This makes the concurrent data structures highly desirable in big data applications such data structures in combination with multi-core machines can be exploited to accelerate the big data applications. But one of the greatest challenges with CDSs is developing correct structures and then proving their correctness either through automatic verification or through hand-written proofs [4]. We believe that the techniques which help to prove correctness of CDSs can also guide in developing new CDSs.

A CDS exports methods which can be invoked concurrently by different threads. A *history* generated by a CDS is a collection of method invocation and response events. Each invocation or *inv* event of a method call has a subsequent response or *rsp* event which can be interleaved with invocation, responses from other concurrent methods.

To prove a concurrent data structure to be correct, *linearizability* proposed by Herlihy & Wing [7] is the standard correctness criterion used. Linearizability ensures that every concurrent execution simulates the behavior of some sequential execution while not actually executing sequentially and hence leveraging on the performance. A history generated by a CDS is linearizable if (1) The *inv* and *rsp* events can be reordered to get a valid sequential history. (2) The generated sequential history satisfies the object's sequential specification. (3) If a *rsp* event precedes an invocation event in the original history, then this should also be preserved in the sequential reordering.

Several techniques have been proposed for proving linearizability: both hand-written based and through automatic verification. One of the intuitive techniques to prove correctness of CDSs is using *Linearization Points* or *LPs*. A LP is an (atomic) event in the execution interval of each method such that the execution of the entire method seems to have taken place in the instant of that event.

Many of techniques for showing linearizability consider lazy linked-list based concurrent set implementation, denoted as *lazy-list*, proposed by Heller at al [5]. This is one of the popular CDSs used for proving correctness due to the intricacies of LPs of its methods in their execution. A particularly interesting scenario with lazy-list being that the LP of an unsuccessful *contains()* method can lie outside the method's code and depend on the LP of a successful concurrent *add()* method if add's LP lies between the *inv* and *rsp* events of the contains() method [5]. Such scenarios can also occur with other CDSs as well.

Vafeiadis et al. [15] hand-crafted one of the earliest proofs of linearizability for lazy-list using the rely-guarantee approach [8] which can be generalized to other CDSs as well. O'Hearn et al. [12] have developed a generic methodology for linearizability by identifying new property known as *Hindsight* lemma. Their technique is non-constructive in nature. Both these techniques don't depend on the notion of LPs.

Recently Lev-Ari et al. [9, 10] proposed a constructive methodology for proving correctness of CDSs. They have developed a very interesting notion of base-points and base-conditions to prove linearizability. Their methodology manually identifies the base conditions, commuting steps, and base point preserving steps and gives a roadmap for proving correctness by writing semi-formal proofs. Their seminal technique, does not depend on the notion of LPs, can help practitioners and researchers from other fields to develop correct CDSs.

*Work in progress

In spite of several such techniques having been proposed for proving linearizability, LPs continue to remain most popular tool for illustrating correctness of CDSs among practitioners. LPs are popular since they seem intuitive and more importantly are constructive in nature. In fact, we believe using the notion of LPs, new CDS can be designed as well. But one of the main challenges with the LP based approach is to identify the correct LPs of a CDS. Identifying the correct LPs can be deceptively wrong in many cases. For instance, it is not obvious to a novice developer that the LP of an unsuccessful *contains* method of lazy-list could be outside the *contains* method (as explained above). In fact in many cases, the LP identified and as a result even the CDS could be wrong.

The problem of proving correctness of CDS (either with or without LPs) has been quite well explored in the verification community in the past few years. Several efficient automatic proving tools and techniques have been developed [11, 1, 14, 18, 19, 3] to address this issue. In fact, many of these tools can also show correctness without the information of LPs. But very little can be gleaned from these techniques to identify the correct LPs of a CDS by a programmer. Nor do they provide any insight to a programmer to develop new CDSs which are correct. The objective of the most of these techniques has been to efficiently automate proving correctness of already developed CDSs.

Considering the complexity of developing a CDS and verifying its correctness, we address the most basic problem of this domain in this paper: **given the set of LPs of a CDS, how to show its correctness?** We have developed a hand-crafted technique of proving correctness of the CDSs by validating it LPs which is inspired by rely-guarantee approach [8, 15]. Our technique can be applied to prove the correctness of several commonly used CDSs developed in literature such as Lock-free Linked based Sets [16], hoh-locking-list [2, 6] , lazy-list [5, 6], Skiplists [17] etc. Our technique will also work for CDSs in which the LPs of a method might lie outside the method such as lazy-list. To show the efficacy of this technique, we show the correctness of lazy-list and hand-over-hand locking list (*hoh-locking-list*) [2, 6].

As observed earlier, identifying the correct LPs is very tricky and erroneous. But since our technique is hand-crafted, we believe that the process of proving correctness might provide insight to identify the correct LPs, if the currently chosen LP is incorrect. We also believe that this technique might also offer the programmer some insight to develop more efficient variants of the CDS.

Contribution. In the proposed technique, we consider executions corresponding to the histories. For a history H, an *execution* E^H is a totally ordered sequence of atomic events which are executed by the threads invoking the methods of the history. Thus an execution starts from an initial *global state* and then goes from one global state to the other as it executes atomic events. With each global state, we associate the notion of *abstract data-structure* or *AbDS*. This represent the state of the CDS if it had executed sequentially.

Our technique starts with assumptions as detailed in Section 3 over the CDS with given LPs. Then for any concurrent history H, we construct a sequential history $CS(H)$ by ordering all the methods of H by their LPs (which are all atomic and hence totally ordered). The details of this construction are described in Section 3.1. Since $CS(H)$ is generated sequentially, it can be seen that it satisfies the sequential-specification of the CDS, hence legal[7](Assumption 1.2).

All the method invocations of CS(H) respect the method ordering of H (Lemma 2). If we can show that all the response events in H and $CS(H)$ are the same then H is linearizable. The proof of this equivalence naturally depends on the properties of the CDS being considered. We have identified a lemma (Lemma 5) as a part of our proof technique, which if shown to be true for all the methods of the CDS, implies linearizability of the CDS. In this lemma, we consider the pre-state of the LP of a method m_i in a history H. As the name suggests, pre-state is the global state of the CDS just before the LP event. This lemma requires that the AbDS in the pre-state to be the result of some sequential execution of the methods of the CDS. Similarly, the AbDS in the post-state of the LP must be as a result of some sequential execution the methods with m_i being the final method in the sequence. We show that if the CDS ensures this equivalence between concurrent and sequential execution then it is linearizable (as detailed in Theorem 8).

The Lemma 5 serves as a template and has to be proved individually for each CDS subjected for testing the correctness. We show that any CDS for which this lemma is true and satisfies our assumptions on the LPs, is linearizable.

Roadmap. In Section 2, we describe the minimal system model that is necessary to understand the proof technique. In Section 3, we give an high level overview of the proof technique. Finally, we conclude in Section 4. The detailed proofs and model can be found in technical report[13].

2 System model

In this paper, we assume that our system consists of finite set of p processors, accessed by a finite set of n threads that run in a completely asynchronous manner and communicate using deterministic shared objects. The threads communicate with each other by invoking higher-level *methods* on the shared objects and obtaining the corresponding responses. Consequently, we make no assumption about the relative speeds of the threads. We also assume that none of these processors and threads fail. We refer to a shared objects as a *concurrent data-structure* or *CDS*.

Events & Methods. We assume that the threads execute atomic *events*. Similar to Lev-Ari et. al.'s work, [10, 9] we assume that

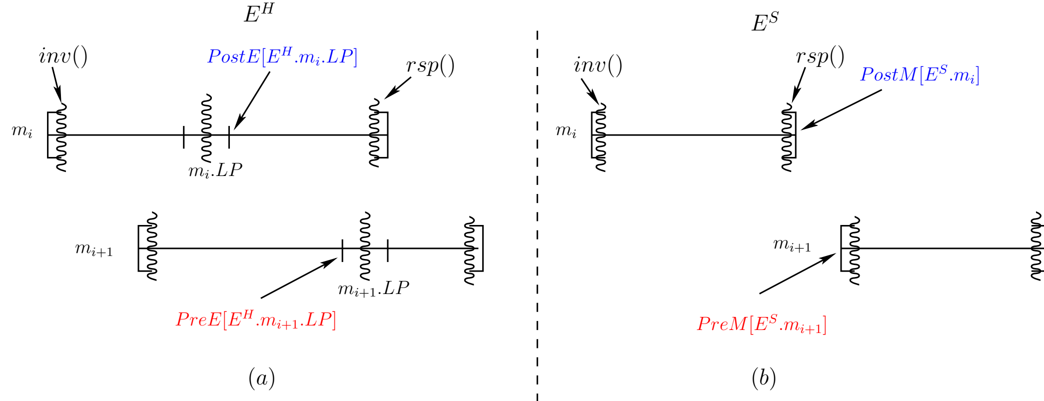

Figure 1: Figure (a) illustrates an example of a concurrent execution E^H. Then, $m_i.LP$ is the LP event of the method m_i. The global state immediately after this event is represented as Post-state of ($E^H.m_i.LP$) and the global state immediately before this event is represented as Pre-state of ($E^H.m_i.LP$). Figure (b) represents sequential execution $E^{\mathbb{S}}$ corresponding to (a) with post-state of method m_i as the state after its rsp event and pre-state of method m_i as the state before its inv event.

these events by different threads are (1) atomic *read, write* on shared/local memory objects; (2) atomic *read-modify-write* or *rmw* operations such compare & swap etc. on CDSs (3) method invocation or inv event & response or rsp event on CDSs.

A thread executing a method m_i, starts with the *inv* event, say inv_i, executes the events in the m_i until the final *rsp* event rsp_i. The *rsp* event rsp_i of m_i is said to *match* the *inv* event inv_i. On the other hand, if the invocation event inv_i does not have a response event rsp_i in the execution, then we say that both the *inv* event inv_i and the method m_i are *pending*. The method *inv* & *rsp* events are typically associated with invocation and response parameters. The invocation parameters are passed as input while response parameters are obtained as output to and from the CDS respectively. We represent the method m_i with invocation and response parameters as: $m_i(inv\text{-}params \downarrow, rsp\text{-}params \uparrow)$ where $inv\text{-}params \downarrow$ are all the parameters passed during invocation and $rsp\text{-}params \uparrow$ are all the parameters returned by the data-structure. In most cases, we ignore these invocation and response parameters unless they are required for the context and denote the method as m_i. In such a case, we simply denote $m_i.inv, m_i.rsp$ as the *inv* and *rsp* events.

Global States, Execution and Histories. We define the *global state* or *state* of the system as the collection of local and shared variables across all the threads in the system. The system starts with an initial global state. Each event possibly changes the global state of the system leading to a new global state.

We denote an *execution* of a concurrent threads as a finite sequence of totally ordered atomic events. A *history* corresponding to an execution consists only of method *inv* and *rsp* events (in other words, a history views the methods as black boxes without going inside the internals). For a history H, we denote the corresponding execution as E^H. We denote the set of methods invoked by threads in a history H (or the corresponding execution E^H) by $H.mths$ (or $E^H.mths$). Similarly, if a method m_x is invoked by a thread in a history H (or E^H), we refer to it as $H.m_x$ (or $E^H.m_x$). Although all the events of an execution are totally ordered in E^H, the methods are only partially ordered. We say that a method m_x is ordered before method m_y in *real-time* if the *rsp* event of m_x precedes the inv event of m_y, i.e. ($m_x.rsp <_H m_y.inv$). We denote the set of all real-time orders between the methods of H by $\prec_H^{rt}$.

Given an event e of an execution E, we denote global state just before the e as the pre-state of e and denote it as $PreE[e]$. Similarly, the state immediately after e as the post-state of e or $PostE[e]$. The notion of pre & post states can be extended to methods as well. We denote the pre-state of a method m or $PreM[m]$ as the global state just before the invocation event of m whereas the post-state of m or $PostM[m]$ as the global state just after the return event of m. Given a method m_i with its LP event as LP, Figure 1 illustrates the global states immediately before and after $m_i.LP$ which are denoted as $PreE[E^H.m_i.LP]$ and $PostE[E^H.m_i.LP]$ respectively in execution E^H.

Notations on Histories. We now define a few notations on histories which can be extended to the corresponding executions. We say two histories $H1$ and $H2$ are *equivalent* if the set of events in $H1$ are the same as $H2$, i.e., $H1.evts = H2.evts$ and denote it as $H1 \approx H2$.

History H as *well-formed* if a thread T_i does not invoke the next method on a CDS until it obtains the matching response for the previous invocation. We assume that all the executions & histories considered in this paper are well-formed. We say the history H is *complete* if for every method *inv* event there is a matching *rsp* event, i.e., there are no pending methods in H. The history H is said to be *sequential* if every *inv* event, except possibly the last, is immediately followed by the matching *rsp* event. In other words,

all the methods of H are totally ordered by real-time and hence $\prec_H^{rt}$ is a total order. Note that a complete history is not sequential and the vice-versa. It can be seen that in a well-formed history H, for every thread T_i, we have that $H|T_i$ is sequential.

Sequential Specification. The *sequential-specification* [7] of a CDS d is defined as the set of (all possible) sequential histories involving the methods of d. Since all the histories in the sequential-specification of d are sequential, this set captures the behavior of d under sequential execution which is believed to be correct. A sequential history $\mathbb{S}$ is said to be *legal* if for every CDS d whose method is invoked in $\mathbb{S}$, $\mathbb{S}|d$ is in the sequential-specification of d.

3 Linearizability Proof Technique

To prove correctness of a CDS d, we associate with it an abstract data-structure or AbDS as explained above. The AbDS captures the behavior of d if it had executed sequentially. The exact definition of AbDS depends on the actual CDS being implemented. In the case of lazy-list, AbDS is the set of unmarked nodes reachable from the head while the CDS is the set of all the nodes in the system. Vafeiadis et. al [15] while proving the correctness of the lazy-list refer to AbDS as *abstract set or AbS*.

Assumption 1 *We make following assumptions for such a CDS d before proceeding to step by step explanation of the proof technique.*

- *1.1 In any sequential execution, any method of the CDS can be invoked in any global state and yet get a response.*
- *1.2 Every sequential history $\mathbb{S}$ generated by the CDS is legal.*
- *1.3 Every method m_i(inv-params $\downarrow$, rsp-params $\uparrow$) of the CDS in E^H has a unique LP event within the* inv *and* rsp *events of m_i for the given inv-params, rsp-params. The LP event can be uniquely identified based on the inv-params, rsp-params of m_i.*
- *1.4 Only the LP events of the methods can change the contents AbDS of the given CDS d.*

Assumption 1.1 refers to *total* methods [6, Chap 10]. It can be seen that the Assumptions 1.3 & 1.4 characterize the LP events. Any event that does not satisfy these assumptions is most likely not an LP. To the best of our knowledge, the assumptions are generic and are satisfied by many of the commonly used CDSs such as Lock-free Linked based Sets [16], hoh-locking-list [2, 6] , lazy-list [5, 6], Skiplists [17] etc. In fact, these assumptions are similar in spirit to the definition of valid LP by Zhu et al [19].

To prove linearizability of a CDS d that satisfies above mentioned assumptions, we have to show that every history generated by d is linearizable. To show this, we consider an arbitrary history H generated by d. For simplicity, we assume that H is complete. From the events of H, we construct a sequential history denoted as $CS(H)$. H is linearizable if (1) $CS(H)$ is equivalent to H; (2) $CS(H)$ respects the real-time order of H and (3) $CS(H)$ is legal. We show how these can be proved using the LPs provided.

3.1 Constructing Sequential History

Given a complete history H consisting of method *inv* & rsp events of a CDS d, we construct $CS(H)$ as follows: Since H is complete, from Assumption 1.3, we can uniquely identify LP event of all the methods. We have a single (hypothetical) thread invoking each method of H (with the same parameters) on d in the order of their LP events. Only after getting the response for the currently invoked method, the thread invokes the next method. From Assumption 1.1, which says that the methods are total, we get that for every method invocation d will issue a response.

Thus we can see that the output of these method invocations is the sequential history $CS(H)$. From Assumption 1.2, we get that $CS(H)$ is legal. The histories H and $CS(H)$ have the same *inv* events for all the methods. But, the *rsp* events could possibly be different. Hence, they may not be equivalent to each other unless we prove otherwise.

For showing H to be linearizable, we further need to show $CS(H)$ is equivalent to H and respects the real-time order H. Now, suppose $CS(H)$ is equivalent to H. Then from the construction of $CS(H)$, it can be seen that $CS(H)$ satisfies the real-time order of H. The following lemma proves it.

Lemma 2 *Consider a history H be a history generated by a CDS d. Let $CS(H)$ be the constructed sequential history. If H is equivalent to $CS(H)$ then $CS(H)$ respects the real-time order of H. Formally, $\langle \forall H : (H \approx CS(H)) \implies (\prec_H^{rt} \subseteq \prec_{CS(H)}^{rt}) \rangle$.*

Now it remains to prove that H is equivalent to $CS(H)$ for showing linearizability of H. But this proof depends on the properties of the CDS d being implemented and is specific to d. Now we give a generic outline for proving the equivalence between H and $CS(H)$ for any CDS.

3.2 Details of the Generic Proof Technique

As discussed above, to prove the correctness of a concurrent (& complete) history H representing an execution of a CDS d, it is sufficient to show that H is equivalent to $CS(H)$. To show this, we have developed a generic proof technique.

It can be obviously seen that to prove the correctness, this proof depends on the properties of the CDS d being considered. To this end, we have identified a CDS-specific lemma which captures the properties required of the CDS d. Proving this CDS-specific lemma for each CDS would imply equivalence of H between $CS(H)$ and hence linearizability of the CDS. To achieve this, we first prove a set of CDS-independent lemmas which would lead us to the CDS-specific lemma.

In the following lemmas, we assume that all the histories and execution considered here are generated from the CDS d. The CDS d satisfies the Assumptions 1.1, 1.2, 1.3, 1.4. Since we are only considering CDS d, we refer to its abstract data-structure as $AbDS$ and refer to its state in a global state S as $S.AbDS$. We enumerate all the methods of a sequential history $\mathbb{S}$ as: $m_1, m_2...m_n$. All the methods of a concurrent history H are enumerated as $m_1, m_2...m_n$ based on the order of their LPs.

Lemma 3 *Consider a sequential execution $E^{\mathbb{S}}$ of the methods of d. Then, the contents of AbDS in the global state after the* rsp *event of m_x is the same as the AbDS before the* inv *event of the consecutive method m_{x+1}. Formally, $\langle \forall m_x \in E^{\mathbb{S}}.mths : PostM[E^{\mathbb{S}}.m_x].AbDS = PreM[E^{\mathbb{S}}.m_{x+1}].AbDS \rangle$.*

Lemma 4 *Consider a concurrent execution E^H of the methods of d. Then, the contents of AbDS in the post-state of LP of m_x is the same as the AbDS in pre-state of the next LP belonging to m_{x+1}. Formally, $\langle \forall m_x \in E^H.mths : PostE[E^H.m_x.LP].AbDS = PreE[E^H.m_{x+1}.LP].AbDS \rangle$.*

Lemma 3 can be observed from the definition of Sequential Execution. Also, from Assumption 1.4, we know that any event between the post-state of $m_i.LP$ and the pre-state of $m_{i+1}.LP$ will not change the $AbDS$. Hence we get this Lemma 4. Now, we formulate equivalence between sequential and concurrent execution (E^S and E^H) using Lemma 5. It can be considered to be a template for each CDS. Based on the CDS involved, this lemma has to be appropriately proved.

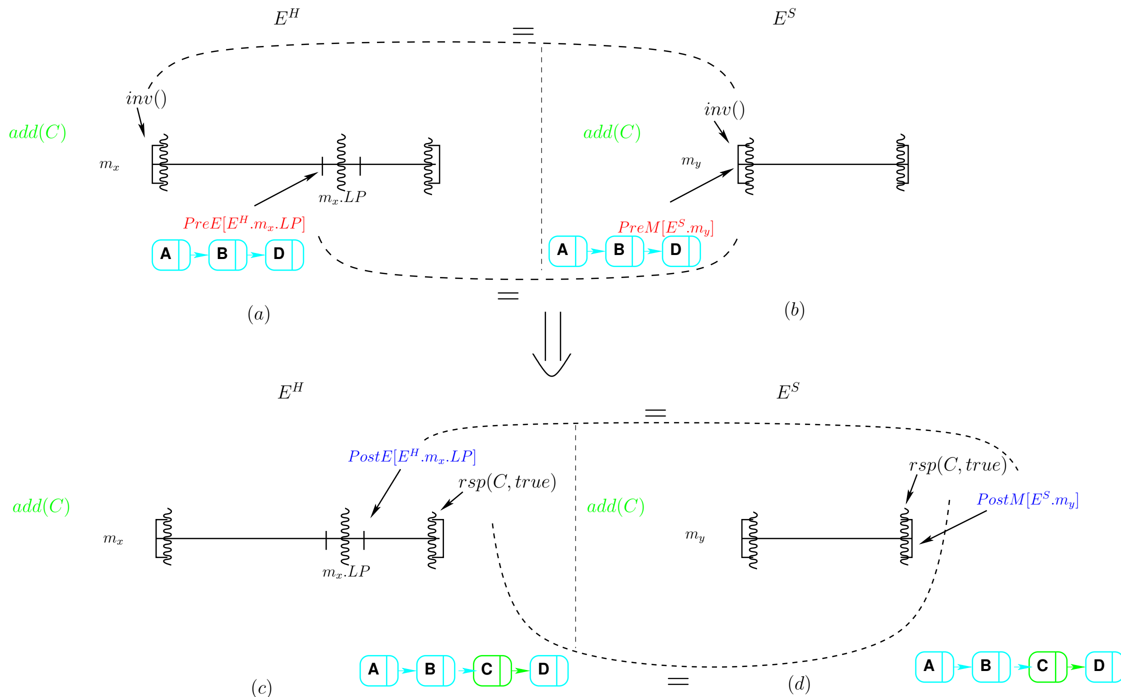

Figure 2: The pictorial representation of the CDS specific lemma over the lazy list. Assume method $add(C)$ executes over the initial list A, B, D. Figure (a) & (b) represent the same *inv* and pre-state for $add(C)$ in concurrent and sequential execution respectively. Then for $add(C)$ execution to be correct its respective post-state and *rsp* should be same in concurrent and sequential executions as depicted in Figure (c) & (d). Note, wlog $add(C)$ is m_x and m_y in E^H & E^S respectively.

CDS-Specific Lemma 5 *Consider a concurrent history H and a sequential history $\mathbb{S}$. Let m_x, m_y be methods in H and $\mathbb{S}$ respectively. Suppose the following are true (1) The AbDS in the pre-state of m_x's LP in H is the same as the AbDS in the pre-state of m_y in $\mathbb{S}$; (2) The* inv *events of m_x and m_y are the same. Then (1) the* rsp *event of m_x in H must be same as* rsp *event of*

m_y *in* $\mathbb{S}$*; (2) The AbDS in the post-state of* m_x*'s LP in* H *must be the same as the AbDS in the post-state of* m_y *in* $\mathbb{S}$*. Formally,* $\langle \forall m_x \in E^H.mths, \forall m_y \in E^{\mathbb{S}}.mths : (PreE[E^H.m_x.LP].AbDS = PreM[E^{\mathbb{S}}.m_y].AbDS) \wedge (E^H.m_x.inv = E^{\mathbb{S}}.m_y.inv) \implies (PostE[E^H.m_x.LP].AbDS = PostM[E^{\mathbb{S}}.m_y].AbDS) \wedge (E^H.m_x.rsp = E^{\mathbb{S}}.m_y.rsp)\rangle$*.*

Figure 2 pictorially represents the statement of the Lemma 5. Readers familiar with the work of Zhu et. al [19] can see that this lemma is similar to Theorem 1 on showing linearizability of CDS d. We prove this lemma specifically for lazy-list and hoh-locking-list.

Next, in the following lemmas we consider the methods of H and $CS(H)$. As observed in Section 3.1, for any method m_x in $CS(H)$ there is a corresponding method m_x in H having the same *inv* event, i.e., $H.m_x.inv = CS(H).m_x.inv$. We use this observation in the following lemmas the detailed proof can be read in technical report[13].

Lemma 6 *For any method* m_x *in* H *and* $CS(H)$ *the AbDS in the pre-state of the LP of* m_x *in* H *is the same as the AbDS in the pre-state of* m_x *in* $CS(H)$*. Formally,* $\langle \forall m_x \in E^H.mths, E^{CS(H)}.mths : PreE[E^H.m_x.LP].AbDS = PreM[E^{CS(H)}.m_x].AbDS\rangle$*.*

Lemma 7 *The return values for all the methods in* H *&* $CS(H)$ *are the same. Formally,* $\langle \forall m_x \in E^H.mths, E^{CS(H)}.mths : E^H.m_x.rsp = E^{CS(H)}.m_x.rsp\rangle$*.*

Theorem 8 *All histories* H *generated by the CDS* d *are linearizable.*

Proof. From Lemma 7, we get that for all the methods m_x, the *rsp* events in H and $CS(H)$ are the same. This implies that H and $CS(H)$ are equivalent to each other. Combining this with Lemma 2, we get that $CS(H)$ respects the real-time order of H. We had already observed that from the construction of $CS(H)$ and Assumption 1.2, $CS(H)$ is legal. Hence H is *linearizable*. □

Analysis of the Proof Technique: Theorem 8 shows that proving CDS specific Lemma 5 implies that the CDS d under consideration is linearizable. Lemma 5 states that for a concurrent execution the contents of the AbDS in the pre-state of the LP event of a method m_x should be the same as the pre-state of the method of some sequential execution. Thus, if the contents of AbDS in the pre-state of the LP event (satisfying the assumptions 1.3 & 1.4) cannot be produced by some sequential execution of the methods of d then it is most likely the case that design of algorithm of the d is incorrect implying LPs may also be faulty.

Further Lemma 5 requires that the AbDS after the execution of the LP of m_x in E^H, must again be same as the AbDS after the method m_x in some sequential execution of of d. If this is not the case, then it implies that some other events of the m_x, are also modifying the AbDS and hence indicating some error in the analysis.

Extending this thought, we also believe that the intuition gained in proving this lemma for d might give the programmers new insights in the working of the CDS which can result in designing new variants of it having some desirable properties.

4 Conclusion and Future Work

In this paper we address the problem: given the set of LPs of a CDS, how to show its correctness? We have developed a hand-crafted technique of proving correctness of the CDSs by validating it LPs. We believe that our technique can be applied to prove the correctness of several commonly used CDSs developed in literature such as Lock-free Linked based Sets [16], lazy-list [5, 6], Skiplists [17] etc. Our technique will also work for CDSs in which the LPs of a method might lie outside the method such as lazy-list. To show the efficacy of this technique, we show the correctness of lazy-list and hand-over-hand locking list (*hoh-locking-list*) [2, 6].

An important point to be noted with our approach: we assumed that only LP events change the AbDS (Assumption 1.4). Although this is true in case of many CDSs considered, this is not always true. As an example consider a shared array which has an lock for each entry and is modified by multiple threads concurrently. Threads wishing to update several entries in a linearizable manner can obtain locks on the relevant entries of the array using two-phase locking (2PL) and then perform the updates. In this case, one can choose any event between the last locking and the first unlocking as the LP. But then, the LP event is not where all the updates to the shared entries of the array takes place. So with this kind of 2PL usage, our technique will not directly work. In that case, we believe that we have to consider the notion of *Linearization Blocks* instead of Linearization Points. We plan to explore this notion in future. On the other hand, we believe that our technique will work for those CDSs which has at least one wait-free method (like the contains method in the case of lazy-list).

References

[1] Daphna Amit, Noam Rinetzky, Thomas W. Reps, Mooly Sagiv, and Eran Yahav. Comparison under abstraction for verifying linearizability. In *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 477–490. Springer, 2007.

[2] Rudolf Bayer and Mario Schkolnick. Concurrency of operations on b-trees. *Acta Informatica*, 9:1–21, 1977.

[3] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. Tractable refinement checking for concurrent objects. In *POPL*, pages 651–662. ACM, 2015.

[4] John Derrick, Gerhard Schellhorn, and Heike Wehrheim. Verifying linearisability with potential linearisation points. In *Proceedings of the 17th International Conference on Formal Methods*, FM'11, pages 323–337, Berlin, Heidelberg, 2011. Springer-Verlag.

[5] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. *Parallel Processing Letters*, 17(4):411–424, 2007.

[6] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

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

[8] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[9] Kfir Lev-Ari, Gregory V. Chockler, and Idit Keidar. On correctness of data structures under reads-write concurrency. In *DISC*, volume 8784 of *Lecture Notes in Computer Science*, pages 273–287. Springer, 2014.

[10] Kfir Lev-Ari, Gregory V. Chockler, and Idit Keidar. A constructive approach for proving data structures' linearizability. In *DISC*, volume 9363 of *Lecture Notes in Computer Science*, pages 356–370. Springer, 2015.

[11] Yang Liu, Wei Chen, Yanhong A. Liu, and Jun Sun. Model checking linearizability via refinement. In *FM*, volume 5850 of *Lecture Notes in Computer Science*, pages 321–337. Springer, 2009.

[12] Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. Verifying linearizability with hindsight. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 85–94, New York, NY, USA, 2010. ACM.

[13] Nandini Singhal, Muktikanta Sa, Ajay Singh, Archit Somani, and Sathya Peri. Proving correctness of concurrent objects by validating linearization points. *CoRR*, abs/1705.02884, 2017.

[14] Viktor Vafeiadis. Automatically proving linearizability. In *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 450–464. Springer, 2010.

[15] Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPOPP*, pages 129–136. ACM, 2006.

[16] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 214–222, New York, NY, USA, 1995. ACM.

[17] V. Luchangco Y. Lev, M. Herlihy and N. Shavit. A provably correct scalable skiplist (brief announcement). In *Proc. of the 10th International Conference On Principles Of Distributed Systems (OPODIS 2006)*, 2006.

[18] Shao Jie Zhang. Scalable automatic linearizability checking. In *ICSE*, pages 1185–1187. ACM, 2011.

[19] He Zhu, Gustavo Petri, and Suresh Jagannathan. Poling: SMT aided linearizability proofs. In *CAV (2)*, volume 9207 of *Lecture Notes in Computer Science*, pages 3–19. Springer, 2015.