Model Checking a Lazy Concurrent List-Based Set Algorithm

Shao Jie Zhang
NUS Graduate School for Integrative Sciences and Engineering
National University of Singapore
Singapore
shaojiezhang@nus.edu.sg

Yang Liu
School of Computing
National University of Singapore
Singapore
liuyang@comp.nus.edu.sg

***Abstract*—Concurrent objects are notoriously difficult to design correctly, and high performance algorithms that make little or no use of locks even more so. In this paper, we present a formal verification of a lazy concurrent list-based set using model checking techniques. The algorithm supports insertion, removal, and membership testing of a list entry under optimistic locking scheme. The algorithm has nonfixed linearization points and is highly non-trivial. We have proved that the algorithm satisfies linearizability, by showing a trace refinement relation from the concrete implementation to its abstract specification. These models are specified in CSP# and verified automatically using our home grown model checker PAT.**

***Keywords*-Linearizability; Concurrent List-Based Set Algorithm; Refinement Checking; PAT**

I. Introduction

Concurrent objects are notoriously difficult to design correctly, and high performance algorithms that make little or no use of locks even more so. The main correctness criterion of the concurrent object design is linearizability [7]. Informally, a shared object is *linearizable* if each operation on the object can be understood as occurring instantaneously at some point, called *linearization point*, between its invocation and its response, and its behavior at that point is consistent with the specification for the corresponding sequential execution of the operation.

Formal verification of linearizability is challenging because the correctness often relies on the knowledge of linearization points, which is difficult or even impossible to identify, not to mention some advanced concurrent algorithms with non-fixed linearization points. These proofs are usually too long and complicated to do (and check) reliably "by hand". Hence, it is important to develop techniques for mechanically performing, or at least checking such proofs.

Heller et al. [6] presented a lazy concurrent list-based set (LCLBS) algorithm. It supports three kinds of list operations: inserting a list entry, removing a list entry and querying whether the list contains a specified entry. All these operations make use of an optimistic locking scheme. That is, query operation uses no locks and thus achieves complete wait-freedom; insertion and removal operations lock the target entry and its predecessor only when it locates the entry, and checks for synchronization conflict. If the conflict is detected, the lock will be released immediately and the operation will be restarted; otherwise, an entry will be successfully added or removed. Recently, Vechev at al. [17] describes a variation of this algorithm with weaker validation condition.

In this paper, we describe a formal verification of the variation in [17] on linearizability using fully automatic model checking techniques. There are three reasons which motivates us to consider this algorithm. Firstly, it is a highly concurrent algorithm with non-fixed linearization points thanks to the optimistic locking scheme. Secondly, every operation requires searching through the list to locate the target entry at first and may change the entry later. The location nondeterminism of a target entry causes the algorithm to become more tricky than most concurrent algorithms. For example, concurrent stacks and queues only perform operations at the endpoints of the underlying data structure. Thirdly, the runtime environment has to enable the algorithm to manipulate dynamic allocated memory heavily and to provide a garbage collector. Such an particular environment is not generally supported by model checking tools. Thus, it arises a challenge of how to achieve memory management during model checking. Therefore, LCLBS algorithm serves as an ideal candidate for automatic verification of concurrent algorithms.

In our approach, the definition of linearizability is cast to the trace refinement relation from concrete implementation model to abstract specification model of an algorithm [9]. Both models are specified using an event-based modeling language CSP# [15] (short for communicating sequential programs), which extends classic Communication Sequential Processes [8] with data variables and low-level programming constructs and shares similar design principle with integrated specification languages like TCOZ [11], [10]. This approach builds on the earlier work in which we proved (and in some cases disproved and/or improved) a number of non-blocking implementations of concurrent stacks, queues and dequeues [9] and scalable nonzero indicator [19]. Basically, an implementation refines a specification if and only if every possible trace of the implementation is one trace in the specification, which indicates the implementation's behaviors conform to the specification. So when satisfying the refinement relation, an implementation is linearizable if the specification is linearizable.

Moreover, in order to model dynamically allocated memory, we pre-allocate an array with a size bound as a private memory space for this algorithm. Allocating space for a list entry and reclaiming the space of an entry are performed upon the array. As for garbage collector, we take advantage of reference counting algorithm. The preallocated memory are stored in the inner library of PAT, and any update and dereference of a list entry are encapsulated as method calls in this library. In this way, the memory management behaves as a black box to the end users and the model remains a clean picture of the algorithm. Furthermore, it is allowable to implement all functions of memory management in C# programming language, which is easier and more convenient than using a high level specification language. The complete model of this list-based set algorithm is built inside a model checking tool, PAT [16], [**?**] (*http://www.patroot.com*).

The rest of the paper is structured as follows. Section II discusses existing works on formal verification of linearizability. Section III presents a brief introduction to LCLBS algorithm. Section IV gives the standard definition of linearizability. Section V shows how to express linearizability using refinement relations in general. Section VI gives the concurrent set algorithm in our modeling language. Section VII presents the verification and experimental results. We end with a conclusion with possible future work in Section VIII.

II. Related Work

Formal verification of linearizability is a much studied research area. There are various approaches in the literature. The idea of refinement has been explored by Alur et al. [1] to show that linearizability can be cast as containment of two regular languages. Derrick et al. [4] proposed the non-atomic refinement based on Object-Z and CSP models. Compared to their work, our definition of linearizability on refinement relation is more general in the sense that it does not require any knowledge of modeling language and linearization points.

Wang and Stoller [18] present a static analysis that verifies linearizability for an unbounded number of threads. Their approach detects certain coding patterns, which are known to be atomic irrespective of the environment. hence is not complete. Amit et al. [2] presented a shape difference abstraction that tracks the difference between two heaps. This approach works well if the concrete heap and the abstract heap have almost identical shapes during the entire algorithm.The main limitation of this approach is that users have to provide linearization points, which is generally unknown. A buggy design may have no linearization points at all. If the linearization points are known, the manual proof is typically sufficient.

```
Entry{
    int key;
    Entry next;
    bool marked;
}
```

Figure 1. Declaration of list entry

To the best of our knowledge, there are two existing works [5] and [17] that verified linearizability for this lazy concurrent list-based set algorithm. Colvin et al. in [5] proved that this algorithm is linearizable by using simulation between input/output automata modeling the behavior of an abstract set and the implementation of a concrete list-based set and then verifying in PVS. However, the theorem prover based approach is not automatic. Conversion to IO automata and use of PVS require strong expertise. Vechev et al. in [17] verified the correctness of this algorithm using SPIN model checker. They provided two methods for linearizability checking. One method requires algorithm-specific user annotation for linearization points. The other is fully automatic. The biggest case they could handle is with two processes and two keys provided user-specified linearization points. The performance of the latter method is unknown. Their model is more intricate than ours and may obscure the understanding of the algorithm itself due to explicitly mixing the code for memory management with their model for the algorithm. Also, much information has to be instrumented into the model to observe actions of other processes for linearizability checking.

III. The List-based Set Algorithm

Three operations supported by the list-based set algorithm should strictly conform to the specification as Table I shows.

The concurrent implementation represents a set as a single-linked sorted list. The list maintains two sentinel nodes *Head* and *Tail*. Every list entry contains three fields as Figure 1 shows. The *key* field denotes a set element. The list is sorted in increasing order of *key*. The *Head* node keeps the minimum possible key and *Tail* node keeps the maximum possible key. The keys in the sentinel nodes can only be read and compared, but not be modified. The *next* field denotes the reference to the next node in the list. Initially, the list only contains *Head* and *Tail*. The *next* field of *Head* points to *Tail* and the *next* field of *Tail* points to *null*. The *marked* field is set to *true* when the entry is about to be removed.

The algorithm is composed of four methods for every process as follows:

Locate is a helper method for three others (see Figure 2). It traverses the list without any locks by following the value of *next* field until *curr* is set to the first entry with a key greater than or equal to "target key". For

Operation	Specification
bool Add (**int** *k*)	Add *k* to the set, succeed and return **true** if *k* was not already in the set; otherwise return **false**
bool Remove (**int** *k*)	Remove *k* from the set , succeed and return **true** if *k* was in the set; otherwise return **false**
bool Contains (**int** *k*)	Succeed and return **true** if *k* is in the set; otherwise return **false**

Table I
SPECIFICATION OF THE LIST-BASED SET ALGORITHM

```
void Locate($pred,\ curr,\ key$){
    $pred = Head$
    $curr = Head \rightarrow next$
    while($curr \rightarrow key < key$){
        $pred = curr$
        $curr = curr \rightarrow next$
    }
}
```

Figure 2. Locate Operation

```
bool Add(int $key$) {
    $Entry\ *pred,\ *curr,\ *entry;$
  restart :
    Locate($pred,\ curr,\ key$)
    $k = (curr \rightarrow key == key)$
    if($k$) return false
    $entry$ = new $Entry(key)$
    $entry \rightarrow next = curr$
    atomic{
        $mp = !(pred \rightarrow marked)$
        $val = (pred \rightarrow next == curr) \&\& mp$
        if($!val$) goto restart
        $pred \rightarrow next = entry$
    }
    return true
}
```

Figure 3. Add Operation

Add operation, "target key" is the one to be inserted; for *Remove* operation, "target key" is the one to be removed; for *Contains* operation, "target key" is the sought-after key. The *pred* entry is set to be the direct predecessor of the *curr*.

Add operation is used to insert a key as Figure 3 shows. It first invokes *Locate* method and gets *curr* and *pred* evaluated. If an entry with the specified key is already in the list, then the method returns false. Otherwise, it creates a new entry with the specified key and evaluates its *next* field to *curr* entry. Then the method locks *curr* and *pred* entries. If *pred* entry is marked to true in the list

```
bool Remove(int $key$) {
    $Entry\ *pred,\ *curr,\ *r;$
  restart :
    Locate($pred,\ curr,\ key$)
    $k = (curr \rightarrow key == key)$
    if($!k$) return false
    $curr \rightarrow marked$ = true
    $r = curr \rightarrow next$
    atomic {
        $mp = !(pred \rightarrow marked)$
        $val = (pred \rightarrow next == curr) \&\& mp$
        if($!val$) goto restart
        $pred \rightarrow next = r$
    }
    return true
```

Figure 4. Remove Operation

or the *next* field of *pred* does not point to *curr*, then the method releases the locks and restarts the insertion process. Otherwise, the new entry is inserted after *pred* entry and the insertion succeeds.

Remove operation is used to remove the entry containing a specified key as Figure 4 shows. Similar to *Add* method, it first invokes *Locate* method. If the list does not contain any entry with the specified key, then the removal fails and returns false. Otherwise, the method marks the *curr* entry to indicate that this entry is logically removed from the list but still in the data structure. Then the method acquires the locks of *pred* and *curr*. If *pred* is already marked or *pred*'s *next* no longer points to *curr*, then the method releases the locks and retries to remove the entry. Otherwise, the *next* field of *pred* is redirected to point to the successor entry, physically removing the entry from the list.

Contains operation is used to test whether the list contains an entry with a specified key as Figure 5 shows. This method invokes *Locate* method just like *Add* and *Remove* method. Instead of locking some entries, the method directly returns true if and only if the *curr* entry contains the desired key.

```
bool Contains(int key){
    Entry *pred, *curr;
    Locate(pred, curr, key)
    k = (curr → key == key)
    if(k) return false
    if(k) return true
```

Figure 5. Contains Operation

IV. Linearizability

Linearizability [7] is a safety property of concurrent systems. It is formalized as follows.

In a shared memory model $\mathcal{M}$, $O = \{o_1, \ldots, o_k\}$ denotes the set of k shared objects, $P = \{p_1, \ldots, p_n\}$ denotes the set of n processes accessing the objects. Shared objects support a set of *operations*, which are pairs of invocations and matching responses. Every shared object has a set of states that it could be in. A *sequential specification* of a (deterministic) shared object is a function that maps every pair of invocation and object state to a pair of response and a new object state.

The behavior of $\mathcal{M}$ is defined as H, the set of all possible sequences of invocations and responses together with the initial states of the objects. A history $\sigma \in H$ induces an irreflexive partial order $<_\sigma$ on operations such that $op_1 <_\sigma op_2$ if the response of op_1 occurs in σ before the invocation of op_2. Operations in σ that are not related by $<_\sigma$ are concurrent. σ is sequential if and only if $<_\sigma$ is a strict total order. Let $\sigma\,|_i$ be the projection of σ on process p_i, which is the subsequence of σ consisting of all invocations and responses that are performed by p_i. Let $\sigma|_{o_i}$ be the projection of σ on object o_i, which consists of all invocations and responses of operations that are performed on object o_i.

A sequential history σ is *legal* if it respects the semantics of the objects as expressed in their sequential specifications. More specifically, for each object o_i, if s_j is the state of o_i before the j-th operation op_j in $\sigma|_{o_i}$, then the invocation and response of op_j and the resulting new state s_{j+1} of o_i follow the sequential specification of o_i. For example, a sequence of read and write operations of an object is legal if each read returns the value of the preceding write if there is one, and otherwise it returns the initial value. Every history σ of a shared memory model $\mathcal{M}$ must satisfy the following basic properties:

- Correct interaction: For each process p_i, $\sigma|_i$ consists of alternating invocations and matching responses, starting with an invocation. This property prevents *pipelining* operations.
- Liveness: Every invocation has a matching response. This property prevents *pending* operations.[1]

Given a history σ, a *sequential permutation* π of σ is a sequential history in which the set of operations as well as the initial states of the objects are the same as in σ. The formal definition of linearizability is given as follows.

Linearizability There exists a sequential permutation π of σ such that 1) for each object o_i, $\pi|_{o_i}$ is a legal sequential history (i.e. π respects the sequential specification of the objects), and 2) if $op_1 <_\sigma op_2$, then $op_1 <_\pi op_2$ (i.e., π respects the real-time ordering of operations).

In every history σ, if we assign increasing time values to all invocations and responses, then every operation can be shrunk to a single time point between its invocation and response such that the operation appears to be completed instantaneously at this time point [3]. This time point for each operation is called its *linearization point*. Linearizability is defined in terms of the invocations and responses of high-level operations, which are implemented by algorithms on concrete shared data structures in real programs. e.g., using a linked list to implement a shared stack object. Therefore, the execution of high-level operations may have complicated interleaving of low-level actions. Linearizability of a concrete concurrent algorithm requires that, despite of complicated low-level interleaving, the history of high-level interface events still has a sequential permutation that respects both the real-time ordering among operations and the sequential specification of the objects. This idea is formally presented in Section V using refinement relations. in a process algebra extended with shared variables.

V. Verification via Refinement Checking

In this section, we first introduce the process algebra extended with shared variables as the modelling language of the concurrent system. After that, labeled transition system generated from the input model is used to describe the behaviors of implementation and specification models. Linearizability is then defined as the refinement relation between the implementation and specification. Lastly, we presents a general algorithm for refinement checking.

A. Modeling Language

We introduce (the relevant subset of) the model language CSP# [15], which is an extension of Communicating Sequential Processes [8] with shared variables, low-level programming constructs and user defined data structures. We choose this language because of its rich set of operators for concurrent communication.

[1]For simplicity, in this paper we do not consider faulty processes, which may have an invocation without a matching response.

Process A process P is defined using the following grammar:

$$\begin{aligned} P ::= {} & Stop \mid Skip \\ \mid {} & e\{program\} \rightarrow P \\ \mid {} & P \setminus X \\ \mid {} & P_1;\ P_2 \\ \mid {} & P_1 \Box P_2 \\ \mid {} & if(b)\ \{P_1\}\ else\ \{P_2\} \\ \mid {} & P_1 \mid\mid\mid P_2 \\ \mid {} & case\{b1 : P_1\ b2 : P_2\ \cdots;\ default : P\} \\ \mid {} & atomic\{P\} \\ e ::= {} & name(.expression)* \end{aligned}$$

where P, P_1, P_2 are processes, e is a name representing an event with an optional sequential program *program*, X is a set of events, and b is a Boolean expression.

Stop is the process that communicates nothing, also called deadlock. $Skip = \checkmark \rightarrow Stop$, where $\checkmark$ is the termination event. Event prefixing $e \rightarrow P$ performs e and afterwards behaves as process P. If e is attached with a sequential program, then the program is executed atomically together with the occurrence of the event. This sequential program could be the modification of shared variables, or method calls of imported C# library classes, etc. Process $P \setminus X$ hides all occurrences of events in X. An event is invisible if it is explicitly hidden by the hiding operator $P \setminus X$. User can also explicitly specify invisible events by using keyword τ. Sequential composition, $P_1;\ P_2$, behaves as P_1 until its termination and then behaves as P_2. External choice $P_1 \Box P_2$ is solved only by the occurrence of an visible event. Conditional choice $if(b)\ \{P_1\}\ else\ \{P_2\}$ behaves as P_1 if the Boolean expression b evaluates to true, and behaves as P_2 otherwise. Regarding *case* process, the condition is evaluated one by one until one which is true is found and then the corresponding process executes. In case no condition is true, the default process will execute. Process $atomic\{P\}$, as its name implies, executes without any intervention of other processes. Indexed interleaving $P_1 \mid\mid\mid P_2$ runs all processes independently except for communication through shared variables. The generalized form of interleaving is written as $\mid\mid\mid x : \{0..n\}@P(x)$. An event may be in a compound form composed of variables and method calls. A process may be recursively defined, and may have parameters (see examples later). The formal syntax and semantics of our language is presented in [15].

In general, it is difficult and inefficient to write complicated functions or advanced data structures in high level modeling languages (e.g. CSP). The most noticeable feature of PAT is that PAT provides an convenient and efficient mechanism to support user defined data types. To make this easier, PAT allows users to define functions and data type in C# language by following the predefined APIs and use them in CSP# models as in Object-Oriented programming language. These C# classes/methods are built as external libraries (in the form of Dynamic Linking Libraries) and loaded during the runtime.

The semantics of a model is defined using a labeled transition system (LTS). Let Σ denote the set of all visible events and τ denote the set of all invisible events. Let Σ^* be the set of finite traces. Let Σ_τ be $\Sigma \cup \tau$. A LTS is a 3-tuple $L = (S, init, T)$ where S is a set of states, $init \in S$ is the initial state, and $T \subseteq S \times \Sigma_\tau \times S$ is a labeled transition relation. Let s, s' be states in S and $e \in \Sigma_\tau$, we write $s \xrightarrow{e} s'$ to denote $(s, e, s') \in T$. We write $s \xrightarrow{e_1, e_2, \cdots, e_n} s'$ iff there exists $s_1, \cdots, s_{n+1} \in S$ such that $s_i \xrightarrow{e_i} s_{i+1}$ for all $1 \leq i \leq n$, $s_1 = s$ and $s_{n+1} = s'$. Let $tr : \Sigma^*$ be a sequence of visible events. $s \overset{tr}{\Rightarrow} s'$ iff there exists $e_1, e_2, \cdots, e_n \in \Sigma_\tau$ such that $s \xrightarrow{e_1, e_2, \cdots, e_n} s'$. The set of traces of L is $traces(L) = \{tr : \Sigma^* \mid \exists s' \in S, init \overset{tr}{\Rightarrow} s'\}$. In this paper, we consider only LTSs with a finite number of states. In particular, we bound the sizes of variable domains by constants, which also bounds the depths of recursions.

Theorem 1 (Refinement): Let $L_{im} = (S_{im}, init_{im}, T_{im})$ be a LTS for an implementation. Let $L_{sp} = (S_{sp}, init_{sp}, T_{sp})$ be a LTS for a specification. L_{im} refines L_{sp}, written as $L_{im} \sqsupseteq_T L_{sp}$, iff $traces(L_{im}) \subseteq traces(L_{sp})$.

B. Linearizability

This section briefly shows how to create high-level linearizable specifications and how to use refinement relation to define linearizability of concurrent implementations.

We define the linearizable specification LTS $L_{sp} = (S_{sp}, init_{sp}, T_{sp})$for a shared object o in the following way. Every execution of an operation of o on a process includes three atomic steps: the invocation action, the linearization action, and the matching response action. The linearization action performs the computation based on the sequential specification of the object. All the invocation and response actions are visible events, while the linearization ones are invisible events. Their complete specification and transition rules in LTS is formally presented in [9]. We now consider a LTS $L_{im} = (S_{im}, init_{im}, T_{im})$ that supposedly implements object o. Theorem 2 characterizes linearizability of the implementation through refinement relations.

Theorem 2: Traces of L_{im} are linearizable iff $L_{im} \sqsupseteq_T L_{sp}$. The proof of theorem 2 is given in [9]. The theorem shows that to verify linearizability of an implementation, it is necessary and sufficient to show that the implementation LTS is a refinement of the specification LTS as we defined above. This provides the theoretical foundation of our verification of linearizability. Notice that the verification by refinement given above does not require identifying low-level actions in the implementation as linearization points, which is the difficult (and sometimes even impossible) task. In fact, the verification can be automatically carried out without any special knowledge about the implementation beyond knowing the implementation code.

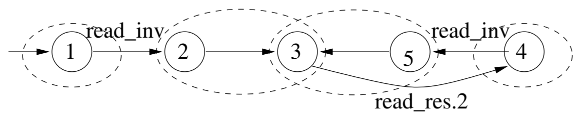

Figure 6. A LTS Example

C. On-the-fly Refinement Checking Algorithm

This section presents a general algorithm for refinement checking, a modified on-the-fly checking algorithm based on the one implemented in FDR. The key idea is to establish a (weak) simulation relationship from the specification to the implementation. Every reachable state of the implementation must be compared with every state of the specification reachable via the same trace. Because of nondeterminism caused by interleaving of multiple clients and invisible events, there may be many such states in the specification. Thus, the specification is firstly normalized, by standard subset construction. A *normalized* state is a set of states that can be reached by the same trace from a given state.

Normalized LTS Let $(S, init, T)$ be a LTS. The normalized LTS is $(NS, Ninit, NT)$ where NS is the set of subsets of S, $Ninit = \tau^*(init)$, and $NT = \{(P, e, Q) \mid Q = \{s : S \mid \exists v_1 : P, \exists v_2 : S\ (v_1, e, v_2) \in T \wedge s \in \tau^*(v_2)\}\}$.

Given a normalized state $s \in NS$, $enabled(s)$ is $\bigcup_{x \in s} enabled(x)$. Given a LTS constructed from a process, the normalized LTS corresponds to the normalized process. A state in the normalized LTS groups a set of states in the original LTS which are all connected by τ-transitions. For instance, the dotted circles in Figure 6 shows the normalized states. Notice that, given a trace, the normalized transition relation NT is deterministic, i.e., for any normalized state P and any event e, there is at most one normalized state Q such that $(P, e, Q) \in NT$. Based on the refinement checking algorithms in FDR [12], we present a modified on-the-fly refinement checking algorithm. Let $Spec = (S_{sp}, init_{sp}, T_{sp})$ be a specification and $Impl = (S_{im}, init_{im}, T_{im})$ be an implementation. Refinement checking is reduced to reachability analysis of the product of *Impl* and normalized *Spec*. Because normalization in general is computationally expensive, our checking algorithm, presented below, performs normalization on-the-fly, whilst searching for a counterexample. By Theorem 2 and the fact that linearizability is a safety property, a counterexample is a finite trace that leads to a state that fails the refinement checking.

The refinement checking algorithm in Figure 7 performs a depth-first search for a pair (Im, NSp) where Im is a state of the implementation and NSp is a normalized state of the specification such that, the set of enabled events of Im is not a subset of those of NSp (C1). The algorithm returns true if no such pair is found. If C1 is satisfied, a counterexample violating trace refinement is found. The

```
procedure $refines(Impl, Spec)$
0.  $checked := \varnothing$;
1.  $pending.push((init_{im}, \tau^*(init_{sp})))$;
2.  while $pending \neq \varnothing$
3.      $(Im, NSp) := pending.pop()$;
4.      $checked := checked \cup \{(Im, NSp)\}$;
5.      if $\neg(enabled(Im) \setminus \{\tau\} \subseteq enabled(NSp))$C1
6.          return $false$;
7.      else
8.          foreach $(Im', NSp') \in \underline{next(Im, NSp)}$
9.              if $(Im', NSp') \notin checked$
10.                 $pending.push((Im', NSp'))$;
11.             endif
12.         endfor
13.     endif
14. endwhile
15. return $true$;
```

Figure 7. Algorithm: $refines(Impl, Spec)$

procedure for producing a counterexample is straightforward and hence omitted. Lines 8 to 12 proceed to explore new states of the product of *Impl* and *Spec* and push them onto the stack *pending*.

Function $next(Im, NSp)$ returns the children of state (Im, NSp) in the product, which is presented in Figure 8. Given a pair (Im, NSp), it returns a set of pairs of the form (Im', NSp') for each enabled event in Im. If the event is visible, NSp' is a successor of NSp via the event and Im' is the successor of Im via the same event. Otherwise, Im' is a successor of Im via a τ-transition and NSp' is Sp. Procedure $tau(S)$ explores all outgoing transition of S and returns the set of states reachable from S via a τ-transition. That is, a new state of the product generated by $next(Im, NSp)$ is obtained by either the implementation taking a silent transition (and the specification remains unchanged) or the implementation and the specification engaging the same event simultaneously.

The checking algorithm is linear in the number of transitions in the product. Assume both *Impl* and *Spec* have finite states. The algorithm terminates because *checked* is monotonically increasing. Because normalization is brought on-the-fly, it is possible to find a counterexample before the specification is completely normalized. The soundness of the algorithm follows the soundness discussion in [12].

VI. List-Based Set Model

In order to prove that the concurrent set algorithm is a linearizable implementation supporting *Add*, *Remove*, *Contains* operations, we model its specification and implementation in extended CSP, and then verify that the implementation refines the specification.

```
procedure $next(Im, NSp)$
0.  $toReturn := \varnothing$
1.  foreach $e \in enabled(Im)$
2.      if $e == \tau$
3.          foreach $Im' \in tau(Im)$
4.              $toReturn := toReturn \cup \{(Im', NSp)\}$;
5.          endfor
6.      else
7.          $NSp' := \{s \mid \exists x : NSp, x \xrightarrow{e} x' \land s \in \tau^*(x')\}$;
8.          foreach $Im'$ such that $Im \xrightarrow{e} Im'$
9.              $toReturn := toReturn \cup \{(Im', NSp')\}$;
10.         endfor
11.     endif
12. endfor
13. return $toReturn$;
```

Figure 8. Algorithm: $next(Im, NSp)$

```
$\#import$ "$PAT.Lib.Set$";
$var\langle Set\rangle\ s = new\ Set()$;

$Sys = P(0,0) ~|||~ \ldots ~|||~ P(N-1,0)$;
$P(i,\ j) = ifa(j < Q)\{$
        $Add(i,\ j) \,\Box\, Remove(i,\ j) \,\Box\, Contains(i,j)$
    $\}$;
$Add(i,\ j) = \Box\, x : \{MIN+1..MAX-1\}@$
        $(add.i.x \rightarrow \tau\{s.Add(x)\} \rightarrow$
         $add.i.x.(s.GetData()) \rightarrow P(i,\ j+1))$;
$Remove(i,\ j) = \Box\, x : \{MIN+1..MAX-1\}@$
        $(rm.i.x \rightarrow \tau\{s.Remove(x)\} \rightarrow$
         $rm.i.x.(s.GetData()) \rightarrow P(i,\ j+1))$;
$Contains(i,\ j) = \Box\, x : \{MIN+1..MAX-1\}@$
        $(ct.i.x \rightarrow \tau \rightarrow ct.i.x.(s.GetData())$
          $\rightarrow P(i,\ j+1))$;
```

Figure 9. Abstract specification model

A. Abstract Specification Model

Figure 9 shows the abstract specification model with N processes. The number of processes and operations, minimum and maximum key values are predefined as constants (*MIN* and *MAX*) for making models finite and subject to model checking. A correct specification model for a set is a fundamental block of refinement checking. However, it is not so straightforward to model such an advanced data structure in high-level specification language like CSP. Therefore, We instead defined a data type *Set* which satisfied the exact specification of a set in Section III as a C# class and built it into PAT's library, so that *Add*, *Remove* and *Contains* operations could be performed as a simple call to the corresponding method of *Set*, such as $s.Remove(x)$ (x is the key of which entry should be removed) in Figure 9.

Every process P could nondeterministically choose an operation among *Add*, *Remove* and *Contains*, and constant Q decides how many operations a process performs. The keyword *ifa* is a special case of conditional choice, which combines condition testing and the first event occurrence in the chosen branch into one atomic step. In this way, we avoid adding unnecessary states which do not exist in the original algorithm. Every operation consist of invocation event, linearization event τ and response event, and nondeterministically selects a key between the minimum and maximin keys. Invocation event contains three parts, respectively indicating which operation, which process and which key. Response event contains four parts: the first three parts have the same meaning as the invocation's; the last one is a method call of *Set*, whose return value is the whole set in an increasing order of key values. It is used to compare with the sorted list, which is the underlying data structure of the set in the implementation model. Thus whether the implementation is consistent with the specification can be decided by the comparison of the set content after executing the same sequence of operations.

B. Concrete Implementation Model

The basic structure of the implementation (the details of every operation are skipped) is showed in Figure 10. Similar to abstract specification model, we define a sorted list entry called *EntryList* which consists of list entries implemented using C# classes and build them into a DLL library. The minimum key *MIN* and maximum key *MAX* are provided to initialize *EntryList* in the object initialization. In order to model dynamically allocated memory, we pre-allocate an array of list entry in *EntryList*. Then, the *next* field of a list entry is modeled as integer indices pointing to this array. M denotes the size of this array. If the length of *EntryList* outnumbers M, then an out-of-memory exception is thrown. In addition, the first element and last element of the array are always reserved as *Head* and *Tail* node of the list.

Having modeled the memory with flat arrays, the next problem is modeling a garbage collector. Here we make use of reference counting algorithm. For this algorithm, the reference counting collector should always keep the number of references to each list entry, i.e. the number of variables whose value is the array index of a list entry. The collector runs whenever the reference number has become zero for some list entries. These entries are deleted, and then the collector runs recursively for all entries pointed to by the *next* field of the deleted entry. The collection stops when no other reachable entry can be deleted. In order to model the reference counting collector, a new integer field *reference* is added to a list entry. It records the number of references to the entry. Whenever a list entry is updated or de-referenced, the *reference* value is modified accordingly. Whenever the *reference* value is decreased to zero, the corresponding entry will be collected, i.e. all of the entry fields are reset to initial values by a method in *EntryList*. The collector always runs atomically. Since the uninitialized or collected entries are supposed to have no influence on the algorithm in practice, only the entry whose *reference* is larger than zero are considered to differentiate the states during refinement checking.

Since PAT does not support local variables (for performance reason), several global arrays are declared to represent local variables of every operation. Taking the *curr* entry as an example, since N processes may perform the identical operations concurrently, and every operation contains *curr* variable, a two dimensional array $curr[N][3]$ is introduced to store *curr* within an operation of P processes. Elements of the same array row denotes *curr* variable in the same operation, i.e. elements of row 0, row 1 and row 2 are used for *Add*, *Remove* and *Contains* operations respectively. Similar are other local variables.

Just like the specification model, a process chooses an operation from *Add*, *Remove* and *Contains* randomly, and every operation chooses a key between the minimum and maximum key values randomly. The invocation and response events of every operation have the same format as the the ones in the specification model. Since every operation invokes *Locate* method firstly, the control of the corresponding CSP process will be immediately transferred to the *Locate* process representing the *Locate* method after the occurrence of invocation event. The operation type is input as a parameter to *Locate* process, so that the control will go back the operation which the type denotes when *Locate* process is finished.

Due to space constraints, we show the resulting code only for *Locate* in Figure 11 and *Add* operation in Figure 12. First, it invokes the *Locate* process (from Line 1 to Line 20). Line 2, 7 and 9 in Figure 11 correspond to the steps of Line 2, 4 and 5 in Figure 2 respectively, all of which involve the reference counter modification of list entries. We define methods to encapsulate the modification details and the operations on reference counters in *EntryList* class. Thus, the original update steps are converted to method calls in the implementation model. Line 25 to 33 and Line 47 to 56 in Figure 12 are the end points of *Add* operation. All local variables are atomically reset along with the response events. As for the variables *pre*, *curr* and *entry* which record the reference to list entries, method *reset* is invoked to decrease the reference counter of these referred entries. Line 35 corresponds to the combination of two process-local actions of Line 7 and 8 in Figure 3.

```
#import "PAT.Lib.EntryList";
//operation types
enum {ADD, REMOVE, CONTAINS};
var<EntryList> l = new EntryList(M, MIN, MAX);
var curr[N][3];
var pred[N][3];
var k[N][3];
var entry[N];
var r[N];
var val[N][2];

Syst = P(0,0) ||| ... ||| P(N − 1,0);
Pro(i, j) = ifa(j < Q){
                (Add(i) □ Remove(i) □ Contains(i));
                Pro(i, j + 1)
            };
Add(i) =   □ x : {MIN + 1..MAX − 1}
         @(add.i.x → Locate(x, i, ADD));
Remove(i) = □ x : {MIN + 1..MAX − 1}
         @(rm.i.x → Locate(x, i, REMOVE));
Contains(i) = □ x : {MIN + 1..MAX − 1}
         @(ct.i.x → Locate(x, i, CONTAINS));
```

Figure 10. Concrete implementation model

VII. Verification and Experimental Result

Based on Theorem 2, automatic refinement checking allows us to verify the linearizability of LCLBS algorithm. PAT [14] supports different notions of refinements based on different semantics. A refinement checking algorithm (inspired by the one implemented in FDR [13]) is used to perform refinement checking on-the-fly. Basically, every reachable state of the implementation must be compared with every state of the specification reachable via the same trace. The key idea is to establish a (weak) simulation relationship from the specification to the implementation. We remark that FDR does not support shared variables/arrays, and therefore, is not easily applicable.

We have experimented LCLBS on PAT for different number of processes and tree nodes. The table below summarizes the results, where Key is the result of $(MAX - MIN)$, '-' means infeasible and '∞' means an unbound number. The testbed is a server with 2.813GHz Intel Xeon 64-bit CPU and 32 GB memory.

Setting			**Result**	
#Proc	**#Key**	**#Operation**	**#States**	**Time(sec)**
2	1	∞	265904	37.06
2	3	1	5867	1.02
2	2	1	23061	3.2
2	2	2	-	-
3	1	1	34979	13.28
3	2	1	-	-

The number of states and running time increase rapidly with data size, and especially the number of processes. This conforms to theoretical results [1]: model checking linearizability is in EXPSPACE for both time and space.

```
Locate(key, p, o) = $\tau$\{pred[p][o] = 0\} $\rightarrow$
          $\tau$\{curr[p][o] = l.GetNext(curr[p][o], 0)\} $\rightarrow$
          l46(key, p, o);

l46(key, p, o) = if (l.Key(curr[p][o]) < key)
{
  $\tau$\{l.Assign(pred[p][o], curr[p][o]);
               pred[p][o] = curr[p][o]; \} $\rightarrow$
  $\tau$\{curr[p][o] = l.GetNext(curr[p][o], curr[p][o])\}
  $\rightarrow$ l46(key, p, o)
}
else
{
   ifa(o == ADD)\{Add5(key, p)\}
   else
    {
      ifa(o == REMOVE) \{Rm22(key, p)\}
      else \{Ct38(key, p)\}
    }
 };
```

Figure 11. Implementation model of *Locate* operation

```
Add5(key, p) =
  $\tau$\{k[p][0] = (l.Key(c[p][0]) == key)\} $\rightarrow$
  if (k[p][0] == true)
  {
    add.p.key.(l.GetData())\{
              k[p][0] = false;
              val[p][0] = false;
              l.Reset(pred[p][0], curr[p][0], entry[p]);
              pred[p][0] = 0;
              curr[p][0] = 0;
              entry[p] = 0;
            \} $\rightarrow$ Skip
  }
  else
  \{$\tau$\{entry[p] = l.Create(entry[p], key, curr[p][0] )\}
    $\rightarrow$
   $\tau$\{val[p][0] =
                 (l.Next(pred[p][0]) == curr[p][0])
                 && !(l.Marked(pred[p][0]));
         if (val[p][0] == true)
         {
             l.SetNext(pred[p][0], entry[p])
         }
     \} $\rightarrow$
    ifa(!val[p][0]) \{Loc(key, p, ADD)\}
    else\{
        add.p.key.(l.GetData())\{
          k[p][0] = false;
          val[p][0] = false;
          Reset(pred[p][0], curr[p][0], entry[p]);
          pred[p][0] = 0;
          curr[p][0] = 0;
          entry[p] = 0
       \} $\rightarrow$ Skip
      \}
  \};
```

Figure 12. Implementation model of *Arrive* operation

Vechev et al. in [17] verified this algorithm in SPIN for two processes and two keys when specifying the linearization points. The performance of their automatic method was not mentioned in their paper. The major drawback of their method is manually inferring linearization points. It is a quite error-prone task of programmers, especially for advanced concurrent algorithms which have no-fixed linearization points. Besides, slight changes to the algorithm can cause the linearization points to change. In contrast, our approach does not require any knowledge of linearization points. Hence, our approach shows more scalability than theirs and is potentially capable of handling all concurrent algorithms.

We have employed several optimization techniques to improve scalability. First, we put the function details relating dynamic memory allocation and reference counting garbage collector into the inner library of PAT, so that no intermediate states during the function execution could be generated. Second, we manually combined sequences of local actions into atomic blocks, such as organizing consecutive events which only cope with local variables into one single τ event. Third, we specified every operation using a minimum number of CSP processes, in order not to generate multiple equivalent states as different parameterized processes containing the same events. Overall, our approach is effective to handle big models like LCLBS.

VIII. Conclusion

In this work, we expressed linearizability using refinement relation. By using this definition, we have successfully verified the lazy concurrent list-based set algorithm. We have shown that the refinement checking algorithm behind PAT allows us to successfully verify complicated concurrent algorithms without the knowledge of linearization points. Moreover, it is shown to be an fairly convenient and efficient way to define new data types and complex functions in a programming language instead of CSP. The fact that new data types and functions are outside the model leaves the model clean and avoid augmenting because of the runtime environment.

During the analysis, we have faced the infamous state explosion problem. In future, we will explore how to combine different state space reduction techniques and parameterized refinement checking for infinite number of processes.

References

[1] R. Alur, K. Mcmillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. In *LICS 96*, pages 219–228. IEEE, 1996.

[2] D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV 07*, pages 477–490. Springer, 2007.

[3] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. John Wiley & Sons, Inc., Publication, 2nd edition, 2004.

[4] J. Derrick, G. Schellhorn, and H. Wehrheim. Proving linearizability via non-atomic refinement. In *IFM 07*, pages 195–214, 2007.

[5] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE 04*, pages 97–114. Springer, 2004.

[6] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N, W. N. S. III, and N. Shavit. A lazy concurrent list-based set algorithm, 2005.

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

[8] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[9] Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model checking linearizability via refinement. In *International Symposium on Formal Methods 2009*, pages 321–337, 2009.

[10] B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An Introduction to TCOZ. In *Proceedings of the 20th International Conference on Software Engineering (ICSE 1998)*, pages 95–104, Kyoto, Japan, 1998.

[11] B. P. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, 2000.

[12] A. W. Roscoe. Model-checking CSP. *A classical mind: essays in honour of C. A. R. Hoare*, pages 353–378, 1994.

[13] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[14] J. Sun, Y. Liu, and J. S. Dong. Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In *ISoLA 08*, pages 307–322. Springer, 2008.

[15] J. Sun, Y. Liu, J. S. Dong, and C. Q. Chen. Integrating Specification and Programs for System Modeling and Verification. In *TASE 09*, 2009.

[16] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV 09*, 2009. (To appear).

[17] M. Vechev, E. Yahav, and G. Yorsh. Experience with model checking linearizability. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*, pages 261–278, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] L. Wang and S. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In *PPoPP 05*, pages 61–71. ACM Press New York, NY, USA, 2005.

[19] S. J. Zhang, Y. Liu, J. Sun, J. S. Dong, W. Chen, and Y. A. Liu. Formal verification of scalable nonzero indicators. In *SEKE 2009*, pages 406–411, 2009.