7-2009

Formal verification of scalable nonzero indicators

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Yang LIU

Jun SUN

Jin Song DONG

Wei CHEN

See next page for additional authors

Citation

SUN, Jun; LIU, Yang; SUN, Jun; DONG, Jin Song; CHEN, Wei; and LIU, Yanhong A.. Formal verification of scalable nonzero indicators. (2009). *Proceedings of the 21st International Conference on Software Engineering & Knowledge Engineering (SEKE'2009), Boston, July 1-3*. 406-411.
Available at: https://ink.library.smu.edu.sg/sis_research/4962

Formal Verification of Scalable NonZero Indicators

Shao Jie Zhang†, Yang Liu†, Jun Sun†, Jin Song Dong†, Wei Chen‡ and Yanhong A. Liu⋆
† National University of Singapore
shaojiezhang@nus.edu.sg, {liuyang,sunj,dongjs}@comp.nus.edu.sg

‡ Microsoft Research Asia
weic@microsoft.com

⋆ State University of New York at Stony Brook
liu@cs.sunysb.edu

Abstract

Concurrent algorithms 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 describe a formal verification of a recent concurrent data structure Scalable NonZero Indicators. The algorithm supports incrementing, decrementing, and querying the shared counter in an efficient and linearizable way without blocking. The algorithm is highly non-trivial and it is challenging to prove the correctness. 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 the model checking toolkit PAT.

1 Introduction

Concurrent algorithms 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 algorithm design is linearizability [6]. 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. These proofs are 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.

In this paper, we present an approach to verify linearizability based on refinement relations between abstract specification and concrete implementation models of a concurrent algorithm. Both are specified using an event-based modeling language, which has formal semantics based on labeled transition systems. We have used this approach to formally verify a recent concurrent algorithm Scalable NonZero Indicators (SNZI) due to Ellen et al. [5], since the algorithm as a complex and useful implementation serves a good candidate for automatic verification. Our approach also builds on earlier work [8] in which we proved (and in some cases disproved and/or improved) a number of concurrent algorithms like nonblocking stacks, nonblocking queues, K-valued Registers and Mailbox problem. We have made considerable progress in understanding how to model algorithms including specifications and implementations to allow model checking to scale up and handle bigger cases. The complete model of SNZI algorithm is built inside a novel model checking tool, PAT [12] (*http://pat.comp.nus.edu.sg*).

The rest of the paper is structured as follows. Section 2 briefly introduces the SNZI algorithm. Section 3 gives the standard definition of linearizability. Section 4 shows how to express linearizability using refinement relations in general. Section 5 gives the SNZI model in our modeling language. Section 6 presents the verification and experimental results. Section 7 discusses related work and concludes.

2 The SNZI Algorithms

A SNZI object behaves similarly to traditional shared counter. It has one shared integer variable *surplus* and supports three operations: *Arrive* increments *surplus* by 1 when a process enters; *Depart* decrements *surplus* by 1 when the process leaves; the only difference from traditional counters is *Query* operation: it returns a boolean value indicating whether the value of *surplus* is greater than 0. We assume that each *Arrive* operation is always followed by a *Depart* operation for the same process. Therefore *surplus* is always greater or equal to 0. The pseudo code in Fig. 1 gives the

```
shared variable : $Surplus$ : integer ;  initially 0
bool Query()  :  return ($Surplus > 0$)
void Arrive() :  $Surplus \leftarrow Surplus + 1$
void Depart() :  $Surplus \leftarrow Surplus - 1$
```

Figure 1. SNZI specification

specification of a SNZI object.

In [5], the authors propose a rooted tree as the underlying data structure of the SNZI objects implementation. An operation on a child node may invoke operations on its parent. An important invariant is used to guarantee the correctness: the *surplus* of parent node is non-zero if and only if there exists at least one child whose *surplus* is non-zero. Thus, if the *surplus* of one node in the tree is non-zero, so does the root. A process begins *Arrive* operation on any node as long as the corresponding *Depart* will be invoked at the same node, and *Query* operation is directly invoked on the root. Every tree node has a counter X that is increased by *Arrive* and decreased by *Depart*. Since the operations on hierarchial nodes differ from those on root node, the algorithms are separated for hierarchical nodes and root node.

The code for hierarchical SNZI nodes is shown in Fig 2. An *Arrive* operation on a hierarchial node invokes *Arrive* operation on its parent node when increasing X from 0 to 1. Otherwise, it completes without invoking any operation. Moreover, a process which increases X from 0 to 1 should firstly set X by an intermediate value $\frac{1}{2}$. Any process which sees $\frac{1}{2}$ must help that process to invoke *parent.Arrive* and try to change X to 1. If a process succeeds in invoking *parent.Arrive* but fails in setting X to 1, it will invoke a compensating *parent.Depart*.

Similarly, a *Depart* operation on a hierarchial node only invokes *Depart* on its parent node when decreasing X from 1 to 0. A version number is added to X to ensure that every change of X will be detected in both *Arrive* and *Depart* operations for hierarchial nodes as well as root node.

The code for root node is shown in Fig 3. In order to reduce frequent accesses to X by *Query*, the solution for the root node separates out an indicator bit I from X. Hence every process can finish *Query* only by reading the bit I. The authors model all accesses to I using *Read*, *Write*, *Load Linked* and *Store Conditional* primitives to tolerate spurious failures when external applications try to modify I.

I is set to true after a 0 to 1 transition of X, and it is unset to false after a 1 to 0 transition of X. Furthermore, an announce bit a is added to X to indicate that I needs to be set. Similar to the intermediate value $\frac{1}{2}$, a process should set a during a 0 to 1 transition and clean it after setting I successfully. Any other process will also set I if it sees that a is set. Once the indicator is set, it can safely clear a to prevent unnecessary future writes to the indicator.

```
shared variables:
 $X = (c, v) : (\mathbb{N} \cup \{\frac{1}{2}\}, \mathbb{N})$; initially$(0,0)$
 parent: scalable indicator
Arrive
  $succ \leftarrow false$
  $undoArr \leftarrow 0$
  while($\neg succ$)
      $x \leftarrow$ Read(X)
      if $x.c \geq 1$ then
          if CAS$(X, x, (x.c+1, x.v))$ then
              $succ \leftarrow$ true
      if $x.c = 0$ then
          if CAS$(X, x, (\frac{1}{2}, x.v+1))$ then
              $succ \leftarrow$ true
              $x \leftarrow (\frac{1}{2}, x.v+1)$
      if $x.c = \frac{1}{2}$ then
          $parent$.Arrive
          if$\neg$CAS$(X, x, (1, x, v))$ then
              $undoArr = undoArr + 1$
  while($undoArr > 0$) do
      $parent$.Depart
      $undoArr = undoArr - 1$
Depart
  while(true) do
      $x \leftarrow$ Read(X)
      if CAS$(X, x, (x.c-1, x.v))$ then
          if $x.c = 1$ then $parent$.Depart
          return
```

Figure 2. Code for hierarchical SNZI node

3 Linearizability

Linearizability [6] 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 iff $<_\sigma$ 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 re-

```
shared variables:
 $X = (c, a, v) : (\mathbb{N}, \text{boolean}, \mathbb{N})$;  initially$(0, \mathit{false}, 0)$
 $I$ : boolean; initially false
Arrive
    repeat
        $x \leftarrow$ Read(X)
        if $x.c = 0$ then $x' \leftarrow (1, \mathit{true}, x.v + 1)$
        else $x' \leftarrow (x.c + 1, x.a, x.v)$
    until CAS$(X, x, x')$
    if $x'.a$ then
        Write$(I$, true$)$
        CAS$(X, x', (x'.c$, false$, x'v))$
Depart
    repeat
1.      $x \leftarrow$ Read(X)
2.      if CAS$(X, x, (x.c - 1$, false$, x.v))$ then
3.          if $x.c \geq 2$ then
4.          repeat
5.              LL$(I)$
6.              if Read$(X).v \neq x.v$ then return
7.              if SC$(I$, false$)$ then return
Query
    return Read$(I)$
```

Figure 3. Code for SNZI root node

sponses 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. 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. 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 4 using refinement relations.

4 Verification via Refinement Checking

We model concurrent systems using a process algebra, whose behavior is described using a labeled transition system. Linearizability is then defined as a refinement relation from an implementation model to a specification model.

4.1 Modeling Language

We introduce the relevant subset of syntax of CSP (Communicating Sequential Processes) [7] extended with shared variables. We choose this language because of its rich set of operators for concurrent communications.

Process A process P is defined using the grammar:

$$P ::= \mathit{Stop} \mid \mathit{Skip} \mid e\{\mathit{program}\} \rightarrow P \mid P \setminus X \mid P_1;\ P_2 \\ \mid P_1 \Box P_2 \mid \mathit{if}(b)\ \{P_1\}\ \mathit{else}\ \{P_2\} \mid P_1 \mid\mid\mid P_2$$

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. $\mathit{Skip} = \checkmark \rightarrow \mathit{Stop}$, 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, the valuation of the shared variables is updated accordingly. For simplicity, assignments are restricted to update only shared variables. Process $P \setminus X$ hides all occurrences of events in X. An event is invisible iff it is explicitly hidden by the hiding operator $P \setminus X$. 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 $\mathit{if}(b)\ \{P_1\}\ \mathit{else}\ \{P_2\}$ behaves as P_1 if the Boolean expression b evaluates to true, and behaves as P_2 otherwise. Indexed interleaving $P_1 \mid\mid\mid P_2$ runs all processes independently except for communication through shared variables. Processes may be recursively defined, and may have parameters (see examples later). The formal syntax and semantics of our language is presented in [11].

To model nonblocking algorithms, our language provides strong support for synchronization primitives, such as *compare − and − swap* (CAS) and *load − linked* (LL)/*store − conditional* (SC), which are elaborated as follows.

CAS[1] The operational semantics of conditional choice requires that the condition evaluation and the first event to be executed of true/false branch be finished in one atomic step. Hence *CAS* primitive can be directly modeled using conditional choices.

```
/ * The pseudo code of CAS semantics * /
bool CAS(ref addr, val exp, val new) :
 atomically {
      if (*addr = exp) {*addr := new; }
      else { }
 }
/ * The CSP representation of CAS * /
if (*addr == old) {τ{*addr = new; } → Skip}
else {Skip}
```

LL/SC[2] In our model, a shared counter *counter* is added to indicate the timestamp when the content of a memory location X is modified and a counter flag is associated with each process. When *LL* is executed by one of the processes, the content of X is read and the value of *counter* is stored in the counter flag. If an external event updates X or the process executes an operation that may invalidate an atomic sequence (e.g., an exception), then *counter* is increased by 1. When the corresponding *SC* is executed, the counter flag is checked. If the flag is equal to *counter*, then *SC* will be successfully executed. Otherwise, nothing can be done.

```
/ * flag[i]     denotes the counter flag of process i * /
LL(i)       = τ{READ X; flags[i] = counter; } → Skip;
SC(i, v)    = if (flags[i] == counter)
                {τ{X = v; counter++; } → Skip}
                else Skip;
Update(v)   = τ{UPDATE X; counter++; } → Skip;
```

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 $traces(L_{im}) \subseteq traces(L_{sp})$.*

4.2 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 [8]. 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 [8]. 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.

5 SNZI Model

In order to prove that SNZI algorithm is a linearizable implementation , we model its specification and implementation in extended CSP, and then verify that the implementation refines the specification.

Fig. 4 shows the abstract specification model with P processes. Process *ArriveA* and *DepartA* consist of invocation event, linearization event τ and response event. Process

[1]CAS atomically compares the content of a memory location to an expected value, and if they are the same, the content of that memory location is assigned to the new given value.

[2]LL/SC are a pair of instructions. *LL* first reads the current content from a memory location X. A subsequent *SC* stores a new value to X only if no updates have happened in between *LL* and *SC*; otherwise, it fails.

```
ArriveA(i)   = arrive_inv.i → τ{surplus++; }
                  → arrive_res.i → Skip;
DepartA(i)   = depart_inv.i → τ{surplus--; }
                  → depart_res.i → Skip;
QueryA()     = query.(surplus > 0) → QueryA();
ProcessA(i)  = ArriveA(i); DepartA(i); ProcessA(i)
SNZIA()      = (||| x : {0..P − 1}@ProcessA(x))\{τ}
                 ||| QueryA();
```

Figure 4. Abstract specification model

```
ArriveI(p, n)  = arrive_inv.p →
                    if(n == 0) ArriveR(p) else Arrive(p, n);
                    arrive_res.p → Skip;
DepartI(p, n)  = depart_inv.p →
                    if(n == 0) DepartR(p) else Depart(p, n);
                    depart_res.p → Skip;
Process(i)     = □ x : {0..N − 1}@
                    (ArriveI(i, x); DepartI(i, x));
Query()        = query.I → Query();
SNZI()         = (||| x : {0..P − 1}@Process(x))\{τ}
                    ||| Query();
```

Figure 5. Concrete implementation model

QueryA recursively reads whether *surplus* is greater than zero or not. *ProcessA* models the behavior of a process, i.e., repeatedly performs an *ArriveA* followed by a *DepartA*. *SNZIA*[3] interleaves all *ProcessA*s and *QueryA* and hides the τ events (i.e., the linearization events).

The basic structure of the implementation (the details of *Arrive* and *Depart* operations are skipped) is showed in Fig. 5. To initialize the rooted tree in the implementation, a size N array named *node* is created to store SNZI objects. The root is $node[0]$, and for $0 < i < N$, the parent of $node[i]$ is $node\lfloor \frac{i-1}{2} \rfloor$. Since P processes may visit the same node concurrently, an $N \times P$ array is introduced to store the local variables within an operation of P processes visiting N nodes. The full implementation model can be found in the built-in examples of PAT [12] (*http://pat.comp.nus.edu.sg*).

A process could visit any node at any time, i.e., which node a process chooses to visit is decided by external environment. Thus, external choice $\Box$ is used to represent a process visiting a node randomly. *ArriveI(p,n)* represents the process p arriving at the node n. If $n = 0$ (the visiting node is the root), then it starts process *ArriveR* which captures how a process enters the root. Otherwise, it starts process *Arrive* which captures how a process arrives a hierarchical node. So does *DepartI*. Due to space constraints, we show the resulting code only for *Depart* operation at the

[3] $||| \; x : \{1..N\}@P(x)$ is same as $P(1) \; ||| \; .. \; ||| \; P(N)$, similarly for $\Box$.

```
1. DepartR(p) =
2.   τ{c[p] = C[0]; a[p] = A; v[p] = V[0]; } →
3.     if(c[p] == C[0] && a[p] == A && v[p] == V[0]){
4.       τ{C[0] = c[p] − 1; A = false; V[0] = v[p]; } →
5.         if(c[p] > 1){τ → Skip}
6.         else{τ → DepartLoop(p)}
7.     }else{τ → DepartR(p)};
8. DepartLoop(p) = τ{counts[p] = count; } →
9.   if(v[p] != V[0]) {τ → Skip}
10.  else{
11.    if(counts[p] != count){τ → DepartLoop(p)}
12.    else{τ{I = false; count++; } → Skip}
13.  };
```

Figure 6. *Depart* operation on root node

root in Fig. 6. The original algorithm of *Depart* includes two-fold loop statements. Each loop is modeled as a recursively defined process. *DepartR* process models the outer loop, while *DepartLoop* models the inner loop. The original X and x are both structured variables composed of three simple variables (represented respectively by (C, A, V) and (c, a, v)). An atomic and invisible event τ containing the assignment statements of c, a and v represents the assignment of x on line 2. Similar is X on line 4. For line 5, 6 and 7, another τ is added between if/else condition and the first event of true/false branch to prevent them from executing in one atomic step. *DepartLoop* contains a pair of *LL*/*SC* primitives. The value of *counter* is recorded when performing *LL* (line 8). Then when the process attempts *SC*, it checks whether the recorded value is same as the current value of *counter* (line 11). If they are not equal, *DepartLoop* is repeatedly invoked (line 11). Otherwise, the process assigns *false* to I and then performs *Skip* event to return control to the invoking process (line 12).

6 Verification and Experimental Result

Based on Theorem 2, automatic refinement checking allows us to verify the linearizability of SNZI algorithm. PAT [10] supports different notions of refinements based on different semantics. A refinement checking algorithm (inspired by the one implemented in FDR [9] but extended with partial order reduction) is used to perform refinement checking on-the-fly. 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. Another candidate tool is the SPIN model checker, which supports verification of LTL properties. Nonetheless, formalization linearizability as LTL formulae results in large LTL formulae and thus not feasible for verification.

We have experimented SNZI on PAT for different number of processes and tree nodes. The table below summarizes the results, where '-' means infeasible, and 'POR' means partial order reduction. The testbed is a PC with 2.83GHz Intel Q9550 CPU and 4 GB memory.

Setting		**Result without POR**		**Result with POR**	
#Proc	**#Node**	**Time(sec)**	**#States**	**Time(sec)**	**#States**
2	2	23.3	28163	17.1	23828
2	3	73.6	62753	41.4	52779
2	4	393	376342	157	173694
2	5	1298	712857	322	341845
2	6	-	-	496	485156
3	2	-	-	6214	8451568

The number of states and running time increase rapidly with data size, and especially the number of processes. This conform to theoretical results [1]: model checking linearizability is in EXPSPACE. We have employed several optimization techniques to improve scalability. First, we use partial order reduction to effectively reduce the search space and running time. 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 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 SNZI.

7 Related Work and Conclusion

The idea of refinement has been explored by Alur, el al. [1] to show that linearizability can be cast as containment of two regular languages. Our definition of linearizability on refinement is more general, regardless of the modeling language and knowledge of linearization points.

Formal verification of linearizability is a much studied research area. There are various approaches in the literature. Verification using theorem provers is another approach [4], where algorithms are proved to be linearizable by using simulation between input/output automata modeling the behavior of an abstract set and the implementation. However, theorem prover based approach is not automatic. Conversion to IO automata and use of PVS require strong expertise. Wang and Stoller [14] present a static analysis that verifies linearizability for an unbounded number of threads. Their approach detects certain coding patterns, hence is not complete (i.e., not applicable to SNZI algorithm). Amit et al. [2] presented a shape difference abstraction that tracks the difference between two heaps. The main limitation of this approach is that users need to provide linearization points, which is generally unknown. A buggy design may have no linearization points at all. In [13], Vechev and Yahav provided two methods for linearizability checking. One method requires user annotations for linearization points. The other is fully automatic but inefficient (The worse case time is exponential in the length of the history). As a result, the number of operations they can check is only 2 or 3. In contrast, our approach handles all possible interleaving of operations given sizes of the shared objects.

In this work, we expressed linearizability using refinement relation. By using this definition, we have successfully verified the SNZI algorithms for the first time. We have shown that the refinement checking algorithm behind PAT allows us to successfully verify complicated concurrent algorithms without the knowledge of linearization points. 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] 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.

[5] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable NonZero Indicators. In *PODC 07*, pages 13–22, 2007.

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

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

[8] Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model Checking Linearizability via Refinement. Technical Report MSR-TR-2009-29, Microsoft Research Asia, March 2009. http://research.microsoft.com/apps/pubs/?id=79938.

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

[10] 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.

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

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

[13] M. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *PLDI 08*, pages 125–135, 2008.

[14] 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.