Model Checking Linearizability via Refinement

Yang Liu[1], Wei Chen[2], Yanhong A. Liu[3], and Jun Sun[1]

[1] National University of Singapore
{liuyang,sunj}@comp.nus.edu.sg
[2] Microsoft Research Asia
weic@microsoft.com
[3] State University of New York at Stony Brook
liu@cs.sunysb.edu

Microsoft Research Technical Report
MSR-TR-2009-29
March 2009

Abstract. Linearizability is an important correctness criteria for implementations of concurrent objects. Automatic checking of linearizability is challenging because it requires that 1) all executions of concurrent operations be serializable, and 2) the serialized executions be correct with respect to the sequential semantics. This paper describes a new method to automatically check linearizability based on refinement relations from abstract specifications to concrete implementations. Our method avoids the often difficult task of identifying linearization points in implementations, but can also take advantage of linearization points if they are given. The method exploits modeling checking of finite state systems specified as concurrent processes with shared variables. Partial order reduction is exploited to effectively reduce the search space. The approach is built into a toolset that supports a rich set of concurrent operators. The tool has been used to automatically check a variety of implementations of concurrent objects, including the first algorithms for the mailbox problem and scalable NonZero indicators. Our system was able to find all known and injected bugs in these implementations.

1 Introduction

Linearizability [9] is an important correctness criterion for implementations of objects shared by concurrent processes, where each process performs a sequence of operations on the shared objects. 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.

One common strategy for proving linearizability of an implementation (used in manual proof or automatic verification) is to identify linearization points in the implementation of all operations and then show that these operations are executed atomically at the linearization points [8, 2, 21]. However, for many concurrent algorithms, it is difficult or even impossible to identify all linearization points. For example, in the K-valued register algorithm (Section 10.2.1 of [4]), linearization points change dynamically depending on the execution history. Therefore, it is desirable to have automatic solutions to verifying these algorithms without knowing linearization points. Nevertheless, existing methods for automatic verification without using linearization points either apply to limited kinds of concurrent algorithms [22] or are unnecessarily inefficient [21].

This paper describes a new method for automatically checking linearizability based on refinement relations from abstract specifications to concrete implementations. Our method does not rely on knowing linearization points, but can take advantage of them if given. The method exploits modeling checking of finite state systems specified as concurrent processes with shared variables, and is not limited to any particular kinds of concurrent algorithms. We exploit powerful optimizations to drastically improve the efficiency and scalability of our checking method.

Refinement requires that the set of execution traces of a concrete implementation be a subset of that of an abstract specification. Thus, we express linearizability as trace refinement of operation invocations and responses from the abstract specification to the concrete implementation, where the abstract specification is correct with respect to sequential semantics. The idea of refinement has been explored before: Alur, el al. [1] showed that linearizability can be cast as containment of two regular languages, and Derrick et al. [6] expressed linearizability as non-atomic refinement of Object-Z and CSP models. We give a general and rigorous definition of linearizability, regardless of the modeling language used, using refinement.

Our model checking method exploits on-the-fly refinement checking (so that counterexamples, if any, can be produced without generating the entire search space, as in FDR [15]), partial order reduction (to effectively reduce the search space), and other optimizations. If linearization points are known and can be marked in the implementation, our approach constructs an even smaller search space. The result is a powerful linearizability checking method that is much more efficient than prior work. A model checking tool, PAT [17] (*http://pat.comp.nus.edu.sg*), is developed to provide automated support to this approach. PAT supports an event-based modeling language that has a rich set of concurrent operators. Our engineering effort realizes all these optimizations in PAT.

We have used PAT to automatically check not only established algorithms, such as concurrent stack and queue algorithms, but also larger and more sophisticated algorithms that were not verified before—the first algorithms for the mailbox problem [3] and scalable NonZero indicators [8]. Both new algorithms use sophisticated data structures and control structures, so the linearization points are difficulty to identify. Counterexamples were reported quickly for incorrect algorithms, such as an incorrect implementation of concurrent queues [16]. Experimental results show that our solution is much more efficient and scalable than prior work [21], as showed in Section 5.

The rest of the paper is structured as follows. Section 2 gives the standard definition of linearizability. Section 3 shows how to express linearizability using refinement relations in general. Section 4 describes verification and optimization methods. Section 5 presents experimental results. Section 6 discusses related work. Section 7 concludes.

2 Linearizability

Linearizability [9] is a safety property of concurrent systems, over sequences of events corresponding to the invocations and responses of the operations on shared objects. 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[1] 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 operation op_1 occurs in σ before the invocation of operation 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 responses that are performed by p_i. Let $\sigma|_{o_i}$ be the projection of σ on object o_i, which is the subsequence of σ consisting 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.[2]

Given a history σ, a *sequential permutation* π of σ is a seqential 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

[1] More rigorously, the sequential specification is for a *type* of shared objects. For simplicity, however, we refer to both actual shared objects and their types interchangibly in the paper.

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

2. if $op_1 <_\sigma op_2$, then $op_1 <_\pi op_2$ (i.e., π respects the real-time ordering of operations).

Linearizability can be equivalently defined as follows: 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 time and response time such that the operation appears to be completed instantaneously at this time point [12, 4]. This time point for each operation is called its *linearization point*. Linearizability is a safety property [12], so its violation can be detected in a finite prefix of the execution history.

Linearizability is defined in terms of the interface (invocations and responses) of high-level operations. In a real concurrent program, the high-level operations are implemented by algorithms on concrete shared data structures, 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 the next section using refinement relations in a process algebra extended with shared variables.

3 Linearizability as Refinement Relations

We model concurrent systems using a process algebra extended with shared variables. The behavior of a model is described using a labeled transition system generated from the model. We define linearizability as a refinement relation from an implementation model to a specification model.

3.1 Process Algebra

We introduce the relevant subset of syntax of CSP (Communicating Sequential Processes) [10] extended with shared variables and give its operational semantics. We chose this language because of its rich set of operators for concurrent communications.

Definition 1 (Process). *A process P is defined using the grammar[3]:*

$$\begin{aligned} P ::= &\ \mathit{Stop} \mid \mathit{Skip} \mid e\{\mathit{assignments}\} \rightarrow P \mid P \setminus X \mid P_1;\ P_2 \mid P_1 \,\Box\, P_2 \\ &\mid P_1 \lhd b \rhd P_2 \mid P_1 \,|||\, P_2 \,|||\, \cdots \,|||\, P_n \end{aligned}$$

where $P, P_1, P_2, \ldots, P_n$ are processes, e is a name representing an event with an optional sequential program $assignments$, X is a set of names, and b is a Boolean expression.

[3] The parallel composition $(P_1 \parallel P_2 \parallel \cdots \parallel P_n)$ is excluded in the paper since it is irrelevant to our discussion. We include it in our technical report [11].

The most noticeable extension to CSP is the use of shared variables for convenience of modeling. It has long been known [10] that one can model a variable as a process parallel to the processes that use it. Nevertheless, direct support of variables allows concise modeling and efficient verification. $Stop$ 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 assignments, 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 $P_1 \lhd b \rhd P_2$ behaves as P_1 if the Boolean expression b evaluates to true, and behaves as P_2 otherwise. Indexed interleaving $P_1 \mid\mid\mid P_2 \mid\mid\mid \cdots \mid\mid\mid P_n$ runs all processes independently except for communication through shared variables. Processes may be recursively defined, and may have parameters (see examples later).

The formal operational semantics is presented in Appendix A. The semantics of a model is defined with a labeled transition system (LTS). Let Σ denote the set of all visible events and τ denote the set of all invisible events. Since invisible events are indistinguishable, we sometimes also use τ to represent an arbitrary invisible event. Let Σ^* be the set of finite traces. Let Σ_τ be $\Sigma \cup \tau$.

Definition 2 (LTS). *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 $(s, e, s') \in T$. The set of enabled events at s is $enabled(s) = \{e : \Sigma_\tau \mid \exists\, s' \in S, s \xrightarrow{e} s'\}$. 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 $s_{n+1} = s'$, and $s \xrightarrow{\tau *} s'$ iff $s = s'$ or $s \xrightarrow{\tau, \cdots, \tau} s'$. The set of states reachable from s by performing zero or more τ transitions is $\tau^*(s) = \{s' : S \mid s \xrightarrow{\tau *} s'\}$. 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 $s \xrightarrow{e_1, e_2, \cdots, e_n} s'$ and $tr = \langle e_1, e_2, \cdots, e_n \rangle \upharpoonright \tau$ is the trace with invisible events removed. The set of traces of L is $traces(L) = \{tr : \Sigma^* \mid \exists\, s' \in S, init \overset{tr}{\Rightarrow} s'\}$.

For example, Fig. 1 shows a LTS[4] generated from $ReaderA$ process in Example 1, where τ labels are omitted for simplicity. Due to the use of shared variables, a state of the system is a pair (P, V), where P is the current process expression, and V is the current valuation of the shared variables represented as a mapping from names to values. Given a LTS $(S, init, T)$, the size of S can be infinite for two reasons. First, variables may have infinite domains. Second, processes may allow unbounded replication by recursion, e.g., $P = (a \rightarrow P;\ c \rightarrow Skip) \Box b \rightarrow Skip$, or $P = a \rightarrow P \mid\mid\mid P$. In this paper, we consider only LTSs with a finite number of states. In particular, we bound the sizes of variable domains by constants. In our examples, bounding the sizes of variable domains also bounds the depths of recursions.

[4] The dotted circles will be explained in Section 4 and could be ignored for now.

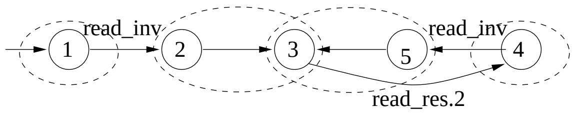

Fig. 1. A LTS Example

Definition 3 (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 $traces(L_{im}) \subseteq traces(L_{sp})$.*

3.2 Linearizability

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

To create a high-level linearizable specification for a shared object, we rely on the idea that in any linearizable history, any operation can be thought of as occurring at some linearization point. We define the specification LTS $L_{sp} = (S_{sp}, init_{sp}, T_{sp})$ for a shared object o in the following way. Every execution of an operation op of o on a process p_i includes three atomic steps: the invocation action $inv(op)_i$, the linearization action $lin(op)_i$, and the response action $res(op, resp)_i$. The linearization action $lin(op)_i$ performs the computation based on the sequential specification of the object. In particular, it maps the invocation and the object state before the operation to a new object state and a response, changes the object to the new state, and buffers the response $resp$ locally. The response action $res(op, resp)_i$ generates the actual response $resp$ using the buffered result from the linearization action. Each of the three actions is executed atomically without being interfered by any other actions, but the three actions of one operation may be interleaved with the actions of other operations. In L_{sp}, all $inv(op)_i$ and $res(op, resp)_i$ are visible events, while $lin(op)_i$ are invisible events.

In a LTS $L_{sp} = (S_{sp}, init_{sp}, T_{sp})$, each process p_i has (a) an idle state $s_{p_i,0}$, (b) a state $s(op)_{p_i,1}$ for every operation op of object o, representing the state after the invocation of op but before the linearization action of op, and (c) $s(op, resp)_{p_i,2}$ for every operation op and every possible response $resp$ of this operation, representing the state after the linearization action of op but before the response of op. Then S_{sp} is the cross product of all object values and all process states. $init_{sp}$ is the combination of the initial value of the object and $s_{p_i,0}$'s for all processes p_i. For $s \in S_{sp}$, let s_{v_o} be the value of object o encoded in s, s_{p_i} be the state of p_i in s, and s_{-p_i} and $s_{-p_i,-v_o}$ be the state of s excluding s_{p_i} and excluding s_{p_i} and s_{v_o}, respectively. The labeled transition relation T_{sp} is such that for $(s, e, s') \in T$, (a) if $e = inv(op)_i$, then $s_{-p_i} = s'_{-p_i}$, $s_{p_i} = s_{p_i,0}$, and $s'_{p_i} = s(op)_{p_i,1}$; (b) if $e = lin(op)_i$, then $s_{-p_i,-v_o} = s'_{-p_i,-v_o}$, $s_{p_i} = s(op)_{p_i,1}$, and $s'_{p_i} = s(op, resp)_{p_i,2}$, such that s'_{v_o} and $resp$ are the new object value and the response, respectively, based on the sequential specification of object o as well as the old object state s_{v_o} and the state $s_{p_i} = s(op)_{p_i,1}$ of process i; (c) if $e = res(op, resp)_i$, then $s_{-p_i} = s'_{-p_i}$, $s_{p_i} = s(op, resp)_{p_i,2}$, and $s'_{p_i} = s_{p_i,0}$.

We now consider a LTS $L_{im} = (S_{im}, init_{im}, T_{im})$ that supposedly implements object o. The visible events of L_{im} are also those $inv(op)_i$'s and $res(op, resp)_i$'s. Theorem 1 characterizes linearizability of the implementation through refinement relations.

Theorem 1. *All traces of L_{im} are linearizable iff $L_{im} \sqsupseteq_T L_{sp}$.*

Proof (sketch). Sufficient condition: For any trace $\sigma \in traces(L_{im})$, because $L_{im} \sqsupseteq_T L_{sp}$, σ is also a trace of L_{sp}. Let ρ be the execution history of L_{sp} that generates the trace σ. We define the sequential permutation π of σ as the reordering of operations in σ in the same order as the linearization actions $lin(op)_i$'s of all operations op and all processes p_i in ρ. If $op_1 <_\sigma op_2$, the linearization action of op_1 must be ordered before the linearization action of op_2 in ρ, and thus $op_1 <_\pi op_2$. It is also easy to verify that π is a legal sequential history of object o, since the linearization action of every operation in ρ is the only action in the operation that affects the object state based on its sequential specification, and the order of operations in π respects the order of linearization actions in ρ.

Necessary condition: Let σ be a trace of L_{im}. By assumption σ is linearizable. We need to show that σ is also a trace of L_{sp}. Since σ is linearizable, there is a sequential permutation π of σ such that π respects both the sequential specification of object o and the real-time ordering of the operations in σ. We construct an execution history ρ of L_{sp} from σ and π as follows. Starting from the first event of σ, for any event e in σ, (a) if it is an invocation event, append it to ρ; (b) if it is a response event $res(op, resp)_i$, locate the operation op in π, and for each unprocessed operation op' by a process j before op in π, process op' by appending a linearization action $lin(op')_j$ to ρ, following the order of π; finally append $lin(op)_i$ and $res(op, resp)_i$ to ρ. It is not difficult to show that the execution history ρ constructed this way is indeed a history of L_{sp}. Moreover, obviously the trace of ρ is σ. Therefore, σ is also a trace of L_{sp}. □

The above 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.

Example 1 (K-valued register). We use a shared K-valued single-reader single-writer register algorithm (Section 10.2.1 of [4]) to demonstrate our idea. The following is the abstract model, where R is the shared register with initial value K, and M is a local variable to store the value read from R.

$$\begin{aligned}
&ReaderA() &&= read_inv \rightarrow read\{M = R;\ \} \rightarrow read_res.M \rightarrow ReaderA();\\
&WriteA(v) &&= write_inv.v \rightarrow write\{R = v;\ \} \rightarrow write_res \rightarrow Skip;\\
&WriterA() &&= (WriteA(0) \,\Box\, WriteA(1) \,\Box \ldots \Box\, WriteA(K-1));\ WriterA();\\
&RegisterA() &&= (ReaderA() \;|||\; WriterA()) \backslash \{read, write\};
\end{aligned}$$

$ReaderA$ process repeatedly reads the value of register R and stores the value in local variable M. $WriteA(v)$ writes the given value v into R. $WriterA$ process repeatedly writes a value in the range of 0 to $K-1$. $RegisterA$ interleaves the reader and writer processes and hides the $read$ and $write$ events (linearization actions).

$$
\begin{array}{ll}
Reader() & = read\_inv \rightarrow UpScan(0); \\
UpScan(i) & = DownScan(i-1, i) \lhd B[i] == 1 \rhd UpScan(i+1); \\
DownScan(i, v) & = (read\_res.v \rightarrow Reader()) \lhd i < 0 \rhd \\
& \qquad (DownScan(i-1, i) \lhd B[i] == 1 \rhd DownScan(i-1, v)); \\
Write(v) & = write\_inv.v \rightarrow \tau\{B[v] = 1;\ \} \rightarrow WriterScan(v-1); \\
WriterScan(i) & = (write\_res \rightarrow Skip) \lhd i < 0 \rhd \\
& \qquad (\tau\{B[i] = 0;\ \} \rightarrow WriterScan(i-1)); \\
Writer() & = (Write(0) \,\Box\, Write(1) \,\Box \ldots \Box\, Write(K));\ Writer(); \\
Register() & = Reader() \;|||\; Writer();
\end{array}
$$

The above models an implementation of a K-valued register using an array B of K binary registers (storing only 0 and 1). The $Reader$ process first does a upward scan from element 0 to the first non-zero element i, and then does a downward scan from element $i-1$ to element 0 and returns the index of last element whose value is 1. The $Write(v)$ process first sets the v-th element of B to 1, and then does a downward scan to set all elements before i to 0. Note that in this implementation, the linearization point for $Reader$ is the last place where the parameter v in $DownScan$ process is assigned. Therefore, the linearization point is not fixed. Instead, it can be in either $UpScan$ or $DownScan$. □

In some cases, one may be able to identify certain events in an implementation as linearization points. We call these linearization events. For example, three linearization events have been identified in the stack algorithm [2]. In these cases, we can make these events visible and hide other events (including the invocation and response events) and verify refinement relation only for these events. More specifically, we obtain a specification LTS L'_{sp} by the following two modifications to L_{sp}: (a) for each linearization action $lin(op)_i$, we change it to $lin(op, resp)_i$ so that the response $resp$ computed by this linearization action is included; and (b) all linearization actions are visible while all $inv(op)_i$ and $res(op, resp)_i$ are invisible. Let L'_{im} be an implementation LTS such that its linearization events are visible and all other events are invisible, and its linearization events are also specified as $lin(op, resp)_i$.

Theorem 2. *Let L'_{sp} and L'_{im} be the specification and implementation LTS such that linearization events are specified as $lin(op, resp)_i$ and are the only visible events. If $L'_{im} \sqsupseteq_T L'_{sp}$, then the implementation is linearizable. Conversely, if the implementation is linearizable, and* it can be shown that no other actions in the implementation can be linearization actions*, then $L'_{im} \sqsupseteq_T L'_{sp}$.*

We skip the proof of the theorem. With this theorem, the verification of linearizability could be more efficient based on only linearization events. However, one important remark is that, as stated in the theorem, to make refinement a necessary condition of linearizability in this case, one has to show that no other actions in the implementation can be linearization points. In other words, the identified linearization points have to be complete. If this is not the case, even if the verification finds a counterexample for the refinement relation, we cannot conclude that the implementation is not linearizable since we may have failed in identifying all possible linearization events. Examples of implementations modeled using linearization points can be found in [11].

4 Verification of Linearizability

This section first presents a general algorithm for refinement checking, extending FDR's algorithm with partial order reduction. We then discuss an optimized way of using this algorithm for linearizability checking. 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.

Definition 4 (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* $NT = \{(P, e, Q) \mid Q = \{s : S \mid \exists\, v_1 : P, \exists\, v_2 : S\ (v_1, e, v_2) \in T \land s \in \tau^*(v_2)\}\}$.

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 1 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 [14], we present a modified on-the-fly refinement checking algorithm that applies partial order reduction. We remark that partial order reduction is an effective reduction method due to the nature of concurrent algorithms. 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 1 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.

procedure $linearizibility(Impl, Spec)$

1. $checked := \varnothing;\ pending.push((init_{im}, \tau^*(init_{sp})));$
2. **while** $pending\ is\ not\ empty$
3. $\quad (Im, NSp) := pending.pop();$
4. $\quad checked := checked \cup \{(Im, NSp)\};$
5. $\quad$ **if** $enabled(Im) \nsubseteq (enabled(NSp) \cup \{\tau\})$ – C1
6. $\qquad$ **return** $false;$
7. $\quad$ **endif**
8. $\quad$ **foreach** $(Im', NSp') \in \underline{next(Im, NSp)}$
9. $\qquad$ **if** $(Im', NSp') \notin checked$
10. $\qquad\quad pending.push((Im', NSp'));$
11. $\qquad$ **endif**
12. $\quad$ **endfor**
13. **endwhile**
14. **return** $true;$

The above algorithm 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 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 the following set,

$$\{(Im', NSp) \mid (Im, \tau, Im') \in T_{im}\} \cup \{(Im', NSp') \mid \exists\, e\ (Im, e, Im') \in T_{im} \wedge \\ \forall\, s : NSp'\ \exists\, s_1 : NSp\, \exists\, s_2 : NSp'\ (s_1, e, s_2) \in T_{sp} \wedge s \in \tau^*(s_2)\}$$

A new state of the product 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. To get $next(Im, NSp)$ (e.g., calculating $\tau^*(s_2)$), it is necessary to compute the set of states reached by a τ-transition from a given state. This function is implemented by procedure $tau(S)$ (Fig. 2, with partial order reduction considerations explained in the next section), which explores all outgoing transitions of S and returns the set of states reachable from S via one τ-transition.

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. The soundness of the algorithm follows from [14]. Because normalization is brought on-the-fly, it is possible to find a counterexample before the specification is completely normalized.

Like any model checking algorithm, linearizability checking suffers from state space explosion. This section describes partial order reduction, which is effective for checking linearizability. Our reduction realizes and extends early works on partial order reduction for process algebras and refinement checking as presented in [20] and [23]. The inspiration of the reduction is that events may be independent, e.g., $read\_inv$ of different readers are mutually independent of each other. Given $P = P_1 \mid\mid\mid \cdots \mid\mid\mid P_n$ and two enabled events e_1 and e_2, e_1 depends on e_2, written as $dep(e_1, e_2)$, if e_1 and e_2

procedure $tau(S)$
foreach P_i
  $por := enabled_{P_i}(S) \subseteq \tau \cup X \wedge enabled_{P_i}(S) = current(P_i)$
  foreach $e \in enabled_{P_i}(S)$
    $por := por \wedge \neg\, onstack(e) \wedge \forall\, e' : \Sigma_j\ j \neq i \Rightarrow \neg\, dep(e, e')$
  endfor
  if por **then**
    return $\{(((\cdots ||| P_i' ||| \cdots) \setminus X), V) \mid (P_i, V) \xrightarrow{e} (P_i', V)\}$;
  endif
endfor
return $\{S' \mid S \xrightarrow{\tau} S'\}$;

Fig. 2. Algorithm: $tau(S)$

are from the same process or e_1 updates a variable to be accessed by e_2, or vice versa. Notice that $dep(e_1, e_2) \Leftrightarrow dep(e_2, e_1)$. Two events are independent if neither depends on the other. Because the ordering of independent events are irrelevant to the truth of linearizability checking, we may ignore some of the ordering so as to reduce the search space. Since interleaving composition is the main source of state space explosion, we consider that Im is in the form of $((P_1 ||| P_2 ||| \cdots ||| P_n) \setminus X, V)$, where P_i is a process, X is a set of events and V is the valuation of the variables. We show how to explore only a subset of enabled transitions and yet preserve soundness.

As discussed above, function $next(Im, NSp)$, which depends on function $tau(S)$, is used to expand the search tree. Therefore, partial order reduction is mainly applied to function $tau(S)$. Because tau is applied to the specification or implementation independently, as long as we guarantee that the reduced state graph (of either $Impl$ or $Spec$) is trace-equivalent to the full state graph, we know that there is a refinement relationship in the reduced state space if and only if there is one in the full state space. Figure 2 shows function $tau(S)$. The idea is to identify one process P_i such that all τ-transitions from P_i are independent of those from other processes, by checking a set of heuristic conditions. Intuitively, a process P_i is chosen if and only if,

- $enabled_{P_i}(S) \subseteq \tau \cup X$, i.e., only invisible transitions are enabled,
- $enabled_{P_i}(S) = current(P_i)$, i.e., all events that could be enabled (given P_i and any valuation of the global variables) are enabled. This is a sufficient condition to guarantee that an event that is dependent on a transition from P_i cannot be executed without a transition from P_i occurring first.
- $\neg\, onstack(e)$, i.e., executing any enabled event does not result in a state on the search stack,
- $\forall\, e' : \Sigma_j,\ j \neq i \Rightarrow \neg\, dep(e, e')$, i.e., all enabled events are independent of events from other processes.

If no such P_i is found, we expand the node with all enabled events. Following the arguments of [20] and [23], it can be shown that the reduced state graph is trace-equivalent to the full graph. The above algorithms apply partial order reduction to τ-transitions only. PAT is capable of applying partial order reduction to visible events. The idea is to identify one event subject to partial order reduction both in Im and every state of NSp. We skip the detail due to space constraint and only remark that reducing visible events is more complicated than reducing τ-transitions (refer to details in [11]).

We remark that by embedding relevant data as part of an event and defining linearizability as refinement relationship, our approach works without knowledge of linearization points. Nonetheless, having the knowledge would allow us to take full advantage of partial order reduction. Because the linearization points are the only places where data consistency must be checked, we may amend the above algorithm to perform data consistency check at the linearization points. As a result, encoding relevant data as part of the event is not necessary and the model contains fewer events, which translates to fewer traces. Furthermore, because only the linearization points need to be synchronized, we may hide all other events, and turn visible transitions into τ-transitions that are subject to partial order reduction.

5 Experiments

Our method has been implemented and applied to a number of concurrent algorithms, including *register*—the k-valued register algorithm in Section 3, *stack*—a concurrent stack algorithm [18], *queue*—a concurrent queue algorithm [13], *buggy queue*—an incorrect queue algorithm [16], and *mailbox* and *SNZI*—the first algorithms for the mailbox problem [3] and scalable Non-Zero indicators [8], respectively. Details for verifying these examples can be found in our technical report [11]. The table below summarizes part of our experiments, where '-' means out of memory or more than 4 hours, 'POR' means partial order reduction, and '(points)' means that linearization points are given. The testbed is a PC with 2.83GHz Intel Q9550 CPU and 4 GB memory.

Algorithm	#Proc.	Linear-izable	Time(sec) w/o POR	#States w/o POR	Time(sec) with POR	#States with POR
4-valued *register*	2	true	6.14	50893	5.72	43977
5-valued *register*	2	true	44.9	349333	60.4	307155
6-valued *register*	2	true	297	2062437	789	1838177
3-valued *register* (2 readers, 1 writer)	3	true	294	479859	393	361255
stack of size 10	2	true	87.7	337577	40.2	243837
stack of size 12	2	true	138	540769	65.9	395345
stack of size 14	2	true	411	763401	99.4	599077
stack of size 2	3	true	-	-	4321	4767519
stack of size 10 (points)	2	true	0.48	6871	0.58	6871
stack of size 12 (points)	2	true	0.62	9677	0.82	9677
stack of size 14 (points)	2	true	0.82	12963	1.11	12963
stack of size 2 (points)	3	true	1.14	10385	1.56	10385
stack of size 2 (points)	4	true	37.6	219471	49.4	219471
queue of size 6	2	true	134	432511	86.2	343446
queue of size 8	2	true	256	104582	218	938542
buggy queue of size 10	2	false	10.9	32126	6.87	32126
buggy queue of size 15	2	false	14.8	63476	14.4	63476
buggy queue of size 20	2	false	52.73	105326	41.1	105326
mailbox of 3 operations	2	true	71.6	272608	27.8	120166
mailbox of 4 operations	2	true	2904	9928706	954	3696700
SNZI of size 3	3	true	-	-	6210	8451568

The number of states and running time increase rapidly with data size and the number of processes, e.g., 3 processes for *register*, *stack*, *queue*, and *SNZI* vs. 2 processes. The results conform to theoretical results [1]: model checking linearizability is in EXPSPACE for both time and space. When linearization points are known, the complexity is still EXPSPACE, but the state space reduces significantly since the state spaces of implementation and specification are smaller. We show that the speedup of knowing linearization points is in the order of $O(2^{k \cdot 2^n \cdot (k^{2n} - k^n)})$, where k is the size of the shared object and n is the number of processes [11]. Use of partial order reduction effectively reduces the search space and running time in most cases, including *stack* and *queue*, and especially *mailbox* and *SNZI* because their algorithms have multiple internal transitions. For *register*, the state space is reduced but running time increases because of computational overhead. For *buggy queue*, the counterexamples in [16] are produced quickly after exploring only part of the state space.

Vechev and Yahav [21] also provided automated verification. Their approach needs to find a linearizable sequence for each history, whose the worst-case time is exponential in the length of the history, as it may have to try all possible permutations 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. Because of partial order reduction and other optimizations, our approach is more scalable than theirs. For instance, we can verify stacks of size 14, i.e., any number of stack operations that contain up to 14 consecutive push operations.

6 Related Work

In terms of modeling of linearizability, our approach is based on the trace refinement of LTSs, which is similar to [1]. The non-atomic refinement defined in [6] separates the data explicitly as state-based formalism Object-Z. Linearization points need to be known in order to create an Object-Z model. Furthermore, this modeling prevents automatic verification techniques such as model checking to be used.

Formal verification of linearizability is a much studied research area, since linearizability is a central property for the correctness of concurrent algorithms. There are various approaches in the literature, as discussed below.

Manual proving Herlihy and Wing [9] present a methodology for verifying linearizability by defining a function that maps every state of an concurrent object to the set of all possible abstract values representing it. Vafeiadis et. al. [19] use rely-guarantee reasoning to verify linearizability for a family of implementations for linked lists. Neither of them requires fixed linearization points, but they are manual.

Using theorem provers Verification using theorem provers (e.g., PVS) is another approach [7, 5]. In these works, 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.

Static analysis Wang and Stoller [22] 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. This solution is not complete (i.e., not applicable to all algorithms). Algorithms such as Michael and Scott's non-blocking queue that do not follow these coding patterns have to be rewritten.

Model checking 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 need to provide linearization points, which is generally unknown. A buggy design may have no linearization points at all. In [21], Vechev and Yahav provided two methods for linearizability checking. The first method is an fully automatic, but inefficient as discussed in Section 5. The second method requires algorithm-specific user annotations for linearization points.

7 Conclusion

In this work, we expressed linearizability using refinement relation. A fully automatic model checking algorithm for linearizability verification is then developed, and is built in a practical tool PAT. Several case studies show that our approach is capable of verifying practical algorithms and identifying bugs from faulty implementations. Several future directions are possible. Algorithms that accept an infinite number of threads or unbound data structures makes model checking impossible. Symmetric properties among threads can reduce infinite number of threads to a small number. Shape analysis can also be incorporated into the model checking to handle unbounded data size.

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. M. K. Arguilera, E. Gafni, and L. Lamport. The mailbox problem. In *DISC'08*, pages 1–15, 2008.
4. H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. John Wiley & Sons, Inc., Publication, 2nd edition, 2004.
5. R. Colvin, S. Doherty, and L. Groves. Verifying Concurrent Data Structures by Simulation. *Electronic Notes in Theoretical Computer Science*, 137(2):93–110, 2005.
6. J. Derrick, G. Schellhorn, and H. Wehrheim. Proving linearizability via non-atomic refinement. In *IFM'07*, pages 195–214, 2007.
7. 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.
8. F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: scalable NonZero indicators. In *PODC'07*, pages 13–22, 2007.
9. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
10. C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985. New version at www.usingcsp.com/cspbook.pdf.
11. Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model Checking Linearizability via Refinement. Technical report, National Univ. of Singapore, 2009. http://www.comp.nus.edu.sg/˜pat/report.ps.
12. N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1997.
13. M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51:1–26, 1998.
14. A. W. Roscoe. Model-checking CSP. *A classical mind: essays in honour of C. A. R. Hoare*, pages 353–378, 1994.
15. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.
16. C. H. Shann, T. L. Huang, and C. Chen. A practical nonblocking queue algorithm using compare-and-swap. In *ICPADS'00*, pages 470–475, 2000.
17. J. Sun, Y. Liu, J. S. Dong, and H. H. Wang. Specifying and verifying event-based fairness enhanced systems. In *ICFEM'08*, pages 318–337, 2008.
18. R. K. Treiber. Systems programming: Coping with parallelism. Technical report, 1986.
19. V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. *PPoPP'06*, pages 129–136, 2006.
20. A. Valmari. Stubborn Set Methods for Process Algebras. In *PMIV'96*, volume 29 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 213–231, 1996.
21. M. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *PLDI'08*, pages 125–135. ACM, 2008.
22. L. Wang and S. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. *PPoPP'05*, pages 61–71, 2005.
23. H. Wehrheim. Partial order reductions for failures refinement. *Electronic Notes in Theoretical Computer Science*, 27, 1999.

Appendix A: Operational Semantics of Extended CSP

The following are Structural Operational Semantics (SOS) rules. We remark that $\Box$, $\sqcap$, $\parallel$ and $|||$ are symmetric and associative. $eval(V, exp)$ evaluates the value of exp given valuation V. Notice that for any P, $\checkmark \in \Sigma_P$.

$$\frac{}{(Skip, V) \xrightarrow{\checkmark} (Stop, V)} [\ skip\ ] \qquad \frac{(P_1, V) \xrightarrow{e} (P_1', V'), e \neq \checkmark}{(P_1 \ ||| \ P_2, V) \xrightarrow{e} (P_1' \ ||| \ P_2, V')} [\ inter\ ]$$

$$\frac{}{(e\{x = exp\} \rightarrow P, V) \xrightarrow{e} (P, V[x/eval(V, exp)])} [\ ass\ ]$$

$$\frac{(P_1, V) \xrightarrow{e} (P_1', V')}{(P_1;\ P_2, V) \xrightarrow{e} (P_1';\ P_2, V')} [\ seq_1\ ] \qquad \frac{(P_1, V) \xrightarrow{\checkmark} (P_1', V')}{(P_1;\ P_2, V) \xrightarrow{\tau} (P_2, V')} [\ seq_2\ ]$$

$$\frac{(P_1, V) \xrightarrow{e} (P_1', V'), e \neq \tau}{(P_1 \Box P_2, V) \xrightarrow{e} (P_1', V')} [\ ex_1\ ] \qquad \frac{(P_1, V) \xrightarrow{\tau} (P_1', V')}{(P_1 \Box P_2, V) \xrightarrow{\tau} (P_1' \Box P_2, V')} [\ ex_2\ ]$$

$$\frac{(P_1, V) \xrightarrow{e} (P_1', V'), V \models b}{(P_1 \lhd b \rhd P_2, V) \xrightarrow{e} (P_1', V')} [\ con_1\ ] \qquad \frac{(P_2, V) \xrightarrow{e} (P_2', V'), V \not\models b}{(P_1 \lhd b \rhd P_2, V) \xrightarrow{e} (P_2', V')} [\ con_2\ ]$$

$$\frac{(P, V) \xrightarrow{e} (P', V'), e \notin X}{(P \setminus X, V) \xrightarrow{e} (P' \setminus X, V')} [\ hid_1\ ] \qquad \frac{(P, V) \xrightarrow{e} (P', V'), e \in X}{(P \setminus X, V) \xrightarrow{\tau} (P' \setminus X, V')} [\ hid_2\ ]$$

$$\frac{(P_1, V) \xrightarrow{e} (P_1', V') \wedge e \notin \Sigma_{P_2}}{(P_1 \parallel P_2, V) \xrightarrow{e} (P_1' \parallel P_2, V')} \qquad \frac{(P_1, V) \xrightarrow{e} (P_1', V) \wedge (P_2, V) \xrightarrow{e} (P_2', V)}{(P_1 \parallel P_2, V) \xrightarrow{e} (P_1' \parallel P_2', V)}$$