1-2013

# Verifying linearizability via optimized refinement checking

Yang LIU

Wei CHEN

Yanhong A. LIU

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

Shao Jie ZHANG

*See next page for additional authors*

## Citation

LIU, Yang; CHEN, Wei; LIU, Yanhong A.; SUN, Jun; ZHANG, Shao Jie; and DONG, Jin Song Dong. Verifying linearizability via optimized refinement checking. (2013). *IEEE Transactions on Software Engineering*. 39, (7), 1018-1039.
**Available at:** https://ink.library.smu.edu.sg/sis_research/4996

# Verifying Linearizability via Optimized Refinement Checking

Yang Liu, Wei Chen, *Member*, *IEEE*, Yanhong A. Liu, Jun Sun, Shao Jie Zhang, and Jin Song Dong

**Abstract**—Linearizability is an important correctness criterion for implementations of concurrent objects. Automatic checking of linearizability is challenging because it requires checking that: 1) All executions of concurrent operations are serializable, and 2) the serialized executions are correct with respect to the sequential semantics. In this work, we describe a method to automatically check linearizability based on refinement relations from abstract specifications to concrete implementations. The method does not require that linearization points in the implementations be given, which is often difficult or impossible. However, the method takes advantage of linearization points if they are given. The method is based on refinement checking of finite-state systems specified as concurrent processes with shared variables. To tackle state space explosion, we develop and apply symmetry reduction, dynamic partial order reduction, and a combination of both for refinement checking. We have built the method into the PAT model checker, and used PAT to automatically check a variety of implementations of concurrent objects, including the first algorithm for scalable nonzero indicators. Our system is able to find all known and injected bugs in these implementations.

**Index Terms**—Linearizability, refinement, model checking, PAT

## 1 Introduction

Linearizability [31] is an important correctness criterion for implementations of objects shared by concurrent processes, each of which 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 the *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 (used in manual proofs and automatic verification) for proving linearizability is to determine linearization points in the implementations of all operations and then show that these operations are executed atomically at the linearization points [20], [3], [58]. However, for many concurrent algorithms (e.g., the elimination back off stack [29], the restricted double-compare single-wrap operation [28], the Herlihy and Wing queue [31], the optimized version [19] of Michael and Scott's lock-free FIFO queue [42], the fine-grained set with wait-free *contains* operation [58], and the scalable nonzero indicators [21]), it is difficult or even impossible to statically determine all linearization points. Taking a particular example, in the K-valued register algorithm [5, Section 10.2.1], linearization points differ depending on the execution history. Furthermore, the linearization points determined might be incorrect, which can lead to wrong verification results. Therefore, it is desirable to have automatic methods for verifying these algorithms without the knowledge of linearization points. However, existing methods for automatic verification without using linearization points either apply to limited kinds of concurrent algorithms [60] or are inefficient [58].

In this work, linearizability is defined as trace refinement of operation invocations and responses from a specification to an implementation, where the specification is correct with respect to sequential semantics. Trace refinement (hereafter refinement) is a subset relationship between traces of two systems. That is, a concrete implementation refines an abstract specification if and only if the set of execution traces of the implementation is a subset of those of the specification. The idea of casting linearizability as refinement has been explored before. Alur et al. [2] showed that linearizability can be cast as containment of two regular languages. Derrick et al. [14] expressed linearizability as nonatomic refinement between Object-Z and CSP models. Other approaches prove linearizability of various algorithms using trace simulation [10], [19], [40].

Our method is not limited to any particular kinds of modeling languages or concurrent algorithms. It exploits model checking of finite state systems specified as concurrent processes with shared variables. In particular, linearizability is verified using refinement checking methods. Though sometimes practically feasible [46], the worst-case execution time of refinement checking is exponential in

- *Y. Liu is with the School of Computer Engineering, Nanyang Technological University, #02C-118, Block N4, 50 Nanyang Avenue, Singapore 639798. E-mail: yangliu@ntu.edu.sg.*
- *W. Chen is with Microsoft Research Asia, Microsoft Asia Pacific R&D Group Headquarters, Building 2, 14-171, 5 Dan Ling Street Haidian District, Beijing 100080, China. E-mail: weic@microsoft.com.*
- *Y.A. Liu is with the Computer Science Department, State University of New York at Stony Brook, Room 1433, Stony Brook, NY 11794-4400. E-mail: liu@cs.sunysb.edu.*
- *J. Sun is with the Singapore University of Technology and Design, 287 Ghim Moh Road, Singapore 279623. E-mail: sunjun@sutd.edu.sg.*
- *S.J. Zhang and J.S. Dong are with the Computer Science Department, School of Computing, National University of Singapore, 13 Computing Drive, Singapore 117417, Republic of Singapore. E-mail: {shaojiezhang, dcsdjs}@nus.edu.sg.*

*Manuscript received 29 Aug. 2011; revised 3 May 2012; accepted 12 Nov. 2012; published online 27 Nov. 2012.*
*Recommended for acceptance by P. Inverardi.*
*For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2011-08-0251.*
*Digital Object Identifier no. 10.1109/TSE.2012.82.*

the size of the abstract specification. To handle real-world concurrent objects, we exploit powerful optimizations to improve the efficiency and scalability of our refinement checking algorithm.

First, our refinement checking explores system behaviors on-the-fly so that a counterexample, if it exists, is produced without generating the entire state space.

Second, we combine state-of-the-art state reduction techniques to combat the state space explosion. The first one is symmetry reduction. Symmetry reduction targets a system composed of sets of behaviorally similar or identical components. Such similarity, or symmetry, often induces equivalent portions of the underlying state space of a system. Provided that the satisfaction of a property to be checked remains unchanged at each equivalent state, exploring one state among the equivalent states is sufficient for verifying the property. A system that models multiple processes manipulating a shared object concurrently tends to exhibit a high degree of symmetry since each operation on this object often originates from a generic system description without distinguishing the processes. Usually either all processes are symmetric, or they can at least be divided into several classes of symmetric processes. For example, in the algorithm for mutual exclusion without priority, each process competing for access to a critical section is equivalent to one another, and thus this system exhibits full symmetry; in the readers-writers protocol, all readers have the same behavior, and so they are "interchangeable," whereas readers and writers cannot be interchanged. Therefore, the readers-writers protocol contains symmetry in readers, but the global behavior is asymmetric. Based on this observation, we apply the symmetry reduction technique for refinement model checking [43] to exploit symmetry between similar processes to reduce the state space.

The second one is partial order reduction. Concurrently, executing processes generate different interleaving traces, and these traces often produce equivalent behaviors. The intuitive idea of partial order reduction is to explore only one interleaving ordering of equivalent traces. In practice, many concurrent object algorithms are designed to minimize the costs of interprocess communication and coordination for scalability reason by reducing the granularity and frequency of locking. Due to the loose coupling between processes, processes potentially have a number of independent steps, and partial order reduction can be fairly efficient. Because pointer variables are frequently used in these algorithms, static approaches fail to accurately detect the independence between program statements. Thus, we apply a dynamic approach called Cartesian partial order reduction [27] in this work.

Then, we combine the above two optimization techniques (which has never been explored before in refinement model checking algorithms) to achieve maximum reduction. We prove the soundness and completeness of our combination algorithm. Experimental results show that the combination of partial order and symmetry can yield even better reductions in model checking concurrent object algorithms than either of the two techniques alone.

Our method does not rely on the knowledge of linearization points, but can take advantage of them if given. If linearization points are given (e.g., marked in the implementation), our method constructs an even smaller search space. Some of the optimization techniques are specialized for linearizability checking while others are general. The result is a powerful linearizability checking method that is much more efficient than our prior work [37].

We extend the PAT model checker [51], [38][1] to support the proposed approach. PAT supports an event-based modeling language [50] that has a rich set of concurrent operators. We apply the proposed method to automatically check finite-state implementations of concurrent object algorithms, such as concurrent counter and queue algorithms, complicated objects with external garbage collector, such as concurrent list-based set [59], as well as sophisticated algorithms—this work is the first published formal verification of scalable nonzero indicators [20] and the mailbox problem[2] [4]. Both algorithms use sophisticated data structures and control structures, and therefore the linearization points are difficult to determine. Counterexamples were reported quickly for incorrect algorithms, such as an incorrect implementation of concurrent queues [47]. Experimental results confirm that our method with the new optimizations is significantly more efficient and scalable than our prior results [37] and other work [58]. Note that our method only verified the finite-state versions of these algorithms.

The rest of the paper is structured as follows: Section 2 presents the definition of linearizability. Section 3 shows how to cast linearizability as refinement relations and proves its correctness. Section 4 describes the verification algorithm and the optimization methods. Section 5 presents experimental results. Section 6 discusses related work. Section 7 concludes.

## 2 LINEARIZABILITY

Linearizability [31] is a safety property of concurrent systems, over sequences of actions corresponding to the invocations and responses of the operations on shared objects. We begin by formally defining the shared memory model.

**Definition 1 (System Models).** *A shared memory model $\mathcal{M}$ is a 3-tuple structure $(O, init_O, P)$, where $O$ is a finite set of shared objects, $init_O$ is the initial valuation of $O$, and $P$ is a finite set of processes accessing the objects.*

Every shared object has a set of states that it could be in. Each shared object supports a set of *operations*, which are pairs of invocations and matching responses. These operations are the only means of reading or writing the state of the object. A shared object is *deterministic* if, given the current state of the object and an invocation of an operation, the next state of the object, as well as the return value of the operation, are unique. Otherwise the shared object is *nondeterministic*. A *sequential specification*[3] of a deterministic (respectively, nondeterministic) shared object is a function that maps every pair of

1. Available at http://www.patroot.com.
2. This algorithm is omitted in this paper due to its size. The details can be found in PAT built-in examples.
3. 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 interchangeably in this paper.

invocation and object state to a pair (respectively, a set of pairs) of response and a new object state.

Formally, an execution of a shared memory model $\mathcal{M} = (O, init_O, P)$ is modeled by a history which is a sequence of operation invocations and response actions that can be performed on $O$ by processes in $P$. The behavior of $\mathcal{M}$ is defined as the set, $H$, of all possible histories together. 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 $\sigma$ before the invocation of operation $op_2$. Operations in $\sigma$ that are not related by $<_\sigma$ are concurrent. A history $\sigma$ is *sequential* iff $<_\sigma$ is a strict total order. Let $\sigma\mid_i$ be the projection of $\sigma$ on process $p_i$, which is the subsequence of $\sigma$ consisting of all invocations and responses that are performed by $p_i$ in $P$. Let $\sigma\mid_{o_i}$ be the projection of $\sigma$ on object $o_i$ in $O$, which is the subsequence of $\sigma$ consisting of all invocations and responses of operations that are performed on object $o_i$. Every history $\sigma$ of a shared memory model $\mathcal{M} = (O, init_O, P)$ must satisfy the following basic properties:

- *Correct interaction:* For each process $p_i$ in $P$, $\sigma\mid_i$ consists of alternating invocations and matching responses, starting with an invocation. This property prevents *pipelining*[4] operations.
- *Closedness:*[5] Every invocation has a matching response. This property prevents *pending* operations.

A sequential history $\sigma$ is *legal* if it respects the sequential specifications of the objects. More specifically, for each object $o_i$, there exists a sequence of states $s_0, s_1, s_2, \ldots$ of object $o_i$ such that $s_0$ is the initial valuation of $o_i$, and for all $j = 1, 2, \ldots$ according to the sequential specification (the function), the $j$th invocation in $\sigma\mid_{o_i}$ together with state $s_{j-1}$ will generate the $j$th response in $\sigma\mid_{o_i}$ and state $s_j$. 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.

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

**Definition 2 (Linearizability).** *Given a model $\mathcal{M} = (O = \{o_1, \ldots, o_k\}, init_O, P = \{p_1, \ldots, p_n\})$. Let $H$ be the behavior of $\mathcal{M}$. $\mathcal{M}$ is linearizable if for any history $\sigma$ in $H$ there exists a sequential permutation $\pi$ of $\sigma$ such that:*

1. *for each object $o_i$ $(1 \leq i \leq k)$, $\pi\mid_{o_i}$ is a legal sequential history, (i.e., $\pi$ respects the sequential specification of the objects), and*
2. *for every $op_1$ and $op_2$ in $\sigma$, if $op_1 <_\sigma op_2$, then $op_1 <_\pi op_2$, (i.e., $\pi$ respects the runtime ordering of operations).*

Linearizability can be equivalently defined as follows: In every history $\sigma$, 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 [40], [5]. This time point is called its *linearization point*.

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., a linked list that implements a shared stack object [53]. Therefore, the execution of high-level operations may have complicated interleaving of low-level actions. Linearizability of a concrete concurrent algorithm requires that, despite complicated low-level interleaving, the history of high-level invocation and response actions still has a sequential permutation that respects both the runtime ordering among operations and the sequential specification of the objects. This idea is formally presented in the next section using refinement relations.

Linearizability is a safety property [40], so its violation can be detected with a finite prefix of an execution history. However, the liveness property is also important for some critical systems which guarantees the progress of the systems. Even if the model satisfies linearizability, it might not progress as desired. For instance, even under a fair scheduler, Treiber's push/pop [53] might never terminate if there is always another concurrent push/pop. This issue is a known property of lock-free or nonblocking algorithms (e.g., lock-free stacks [53], [12] and lock-free queue [19], [11]). It in fact reflects a deliberate design choice to give up guaranteed termination of individual operations in favor of a weaker guarantee of overall progress to obtain an efficient implementation. This suggests that linearizability is just one of many criteria properties for concurrent object design. We remark that liveness properties can be formulated as linear temporal logic (LTL) formulae (an example is given at the end of Example 1) and checked using standard LTL model checkers (with or without the assumption of a fair scheduler [52], [51]). The PAT model checker supports LTL model checking with a number of different fairness assumptions [52], [51].

## 3 Linearizability as Refinement

In this section, we define system behaviors as labeled transition systems (LTSs) and linearizability as a refinement relationship between two system models (or equivalently two LTSs). We propose two ways of constructing the refinement relation for the case where linearization points are not given and the case where they are given, respectively.

### 3.1 Semantic Model

First of all, we introduce LTSs as the semantic models used to capture the behaviors of shared memory models defined by high-level operations or representing real concurrent programs.

**Definition 3 (LTS).** *An LTS is a tuple $L = (S, init, Act, \rightarrow)$ where $S$ is a finite set of states, $init \in S$ is an initial state, $Act$ is a finite set of actions, and $\rightarrow \subseteq S \times Act \times S$ is a labeled transition relation.*

4. Pipelining operations mean that after invoking an operation, a process invokes another (same or different) operation before the response of the first operation.

5. This property is not required in the original definition of linearizability in [31]. However, adding it will not affect the correctness of our result because, by [31, Theorem 2], for a pending invocation in a linearizable history, we can always extend the history to a complete one and preserve linearizability. We include this property to obviate the discussion for pending invocations.

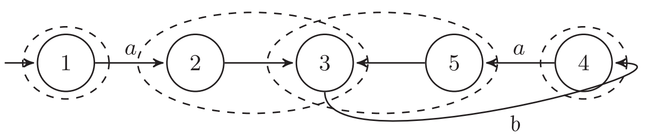

Fig. 1. An LTS example.

For simplicity, we write $s \xrightarrow{\alpha} s'$ to denote $(s, \alpha, s') \in \rightarrow$. The set of enabled actions at $s$ is $enabled(s) = \{\alpha \in Act \mid \exists s' \in S.s \xrightarrow{\alpha} s'\}$. A path $\pi$ of $L$ is a sequence of alternating states and actions, starting and ending with states $\pi = \langle s_0, \alpha_1, s_1, \alpha_2, \ldots \rangle$ such that $s_0 = init$ and $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $i$. If $\pi$ is finite, then $|\pi|$ denotes the number of transitions in $\pi$. A path can also be infinite, i.e., containing an infinite number of actions. Since the number of states are finite, infinite paths are paths containing loops. The set of all possible paths for $L$ is written as $paths(L)$.

A transition label can be either a visible action or an invisible one. Given an LTS $L$, the set of visible actions in $L$ is denoted by $vis_L$ and the set of invisible actions is denoted by $invis_L$. A $\tau$-transition is a transition labeled with an invisible action. A state $s'$ is *reachable* from state $s$ if there exists a path that starts from $s$ and ends with $s'$, denoted by $s \overset{*}{\Rightarrow} s'$. The set of $\tau$-successors is $\tau(s) = \{s' \in S \mid s \xrightarrow{\alpha} s' \wedge \alpha \in invis_L\}$. The set of states reachable from $s$ by performing zero or more $\tau$ transitions, denoted as $\tau^*(s)$, can be obtained by repeatedly computing the $\tau$-successors starting from $s$ until a fixed point is reached. We write $s \xrightarrow{\tau*} s'$ iff $s'$ is reachable from $s$ via only $\tau$-transitions, i.e., there exists a path $\langle s_0, \alpha_1, s_1, \alpha_2, \ldots, s_n \rangle$ such that $s_0 = s$, $s_n = s'$, and $s_i \xrightarrow{\alpha_{i+1}} s_{i+1} \wedge \alpha_{i+1} \in invis_L$ for all $i$ . Given a path $\pi$, we can obtain a sequence of visible actions by omitting states and invisible actions. The sequence, denoted as $trace(\pi)$, is a trace of $L$. The set of all traces of $L$ is written as $traces(L) = \{trace(\pi) | \pi \in paths(L)\}$.

LTSs can often be shown graphically, e.g., Fig. 1 shows an example LTS[6] where invisible transition labels are omitted for simplicity. We define the refinement relation between two LTSs, usually called trace refinement, as follows.

**Definition 4 (refinement).** *Let $L_1$ and $L_2$ be two LTSs. $L_1$ refines $L_2$, written as $L_1 \sqsupseteq_T L_2$, iff $traces(L_1) \subseteq traces(L_2)$.*

## 3.2 Linearizability without Linearization Points

In this section, we make no assumption on the knowledge of the linearization points, which can be known or unknown. We show how to create high-level linearizable specifications and how to define linearizability as refinement. Linearizability is a local property, i.e., a system is linearizable iff each individual shared object is linearizable. Hence, we assume there is only one shared object in a system without loss of generality.

### *3.2.1 Linearizable Specification*

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. For a shared object $o$, we define a specification model $\mathcal{M}_{sp} = (\{o\}, init_{\{o\}}, P_{sp})$ as follows: Every execution of an operation $op$ of $o$ on a process $p_i \in P_{sp}$ includes three atomic steps: the invocation action $inv(op)_i^o$, the linearization action $lin(op)_i^o$, and the response action $res(op, resp)_i^o$. Since there is only one object $o$, we omit the superscript $^o$ for simplicity. 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 stores the response $resp$ locally. The response action $res(op, resp)_i$ generates the actual response $resp$ using the stored result from the linearization action. Each of the three actions is executed atomically without interruption. However, the three actions of an operation may be interleaved with actions of other operations being performed by other processes.

6. The dotted circles will be explained in Section 4.

Given a deterministic shared object $o$, the corresponding model $\mathcal{M}_{sp} = \{\{o\}, init_{\{o\}}, P\}$ can be constructed as follows: Each operation $op$ that can be performed by process $p_i \in P$ is defined as a circular state machine with three states 1) an idle state $s_{p_i,0}$, 2) a state $s(op)_{p_i,1}$ after the invocation of $op$ but before the linearization action of $op$, and 3) a state $s(op, resp)_{p_i,2}$ for every response $resp$ of $op$, representing the state after the linearization action of $op$ but before the response of $op$. The transition from state $s_{p_i,0}$ to state $s(op)_{p_i,1}$ is triggered by an invocation action $inv(op)_i$. The transition from state $s(op)_{p_i,1}$ to state $s(op, resp)_{p_i,2}$ is triggered by a linearization action $lin(op)_i$. The transition from state $s(op, resp)_{p_i,2}$ to state $s_{p_i,0}$ is triggered by a response action $res(op, resp)_i$. We let all invocation actions and response actions be visible actions and all linearization actions be invisible actions. This is because only the invocation and response are considered in the behavior of $\mathcal{M}_{sp}$ (refer to Section 2 for the definition of behavior). Each process is defined as the nondeterministic choice of invoking all the allowed operations on object $o$.

Let $L_{sp} = (S_{sp}, init_{sp}, Act_{sp}, \rightarrow_{sp})$ be the semantic model of $\mathcal{M}_{sp}$. Then, $S_{sp}$ is the cross product of all object values and all process states. Initial state $init_{sp}$ is the combination of the initial value of object $o$ 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 $s$ excluding $s_{p_i}$ and excluding $s_{p_i}$ and $s_{v_o}$, respectively. The labeled transition relation $\rightarrow_{sp}$ is such that for $(s, \xrightarrow{e}, s') \in \rightarrow_{sp}$ , 1) if $e = inv(op)_i$, then $s_{-p_i} = s'_{-p_i}$, $s_{p_i} = s_{p_i,0}$, and $s'_{p_i} = s(op)_{p_i,1}$; 2) 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 $s'_{p_i} = s(op, resp)_{p_i,2}$, where $s'_{v_o}$ and $resp$ are the new object value and the response, respectively, based on the sequential specification of object $o$ and the old object state $s_{v_o}$ and the state $s_{p_i} = s(op)_{p_i,1}$ of process $p_i$; 3) if $e = res(op, resp)_i$, then $s_{-p_i} = s'_{-p_i}$, $s_{p_i} = s(op, resp)_{p_i,2}$, and $s'_{p_i} = s_{p_i,0}$.

To create a high-level linearizable specification for a nondeterministic shared object, the same idea above applies. Assume the object $o$ has $j$ nondeterministic response values after invoking operation $op$. Formally, each operation $op$ that can be performed by process $p_i$ is defined as a state machine with $(j + 2)$ states: 1) an idle state $s_{p_i,0}$, 2) a state $s(op)_{p_i,1}$ after the invocation of $op$ but before the linearization action of $op$, and 3) $j$ states $\{s(op, resp_1)_{p_i,2_1} \ldots s(op, resp_j)_{p_i,2_j}\}$ for all possible responses of $op$, representing the states after the linearization action of $op$ but before the response of $op$. Circular state transitions are similarly defined as well.

### 3.2.2 Implementation Formalization

To formalize any given concurrent algorithm, we need to identify the invocation and response actions so that histories can be formulated based on these actions. For a concurrent algorithm, implementing a shared object $o$, we define an implementation model $\mathcal{M}_{im} = (\{o\}, init_{\{o\}}, P_{im})$. We assume that $P_{im}$ is a parallel execution of all processes. Furthermore, the behavior of each process is assumed to be infinite nondeterministic invocations of all the operations supported by the object. In this work, we use $Process(i)$ and $System$ to model the behaviors of process $P_i$ and the algorithm, respectively. One example can be found in Algorithm 3.3. Each operation is defined by the algorithm. Executions of program statements shall be considered as (invisible) actions. The atomicity of a statement execution can be defined based on the actual hardware architectures. For example, if a computer architecture can compute $x = x + 1$ in one step, then we can treat this statement as atomic action under this particularly architecture. Taking another example, if a computer architecture supports an atomic compare-and-swap (CAS[7]) instruction, then executing the statements representing a CAS is an atomic action under this particularly architecture. Further, local statements can be grouped into one atomic action to reduce the state space of LTS. This can be considered as manual partial order reduction. Since we are interested in the histories of the algorithm, an invocation action is added to the beginning of each operation and a response action is added to every return statement of each operation. All other actions (i.e., the statement execution) inside the algorithm are treated as invisible action since they do not contribute to the histories. The semantic model of $\mathcal{M}_{im}$ is denoted by an LTS $L_{im} = (S_{im}, init_{im}, Act_{im}, \rightarrow_{im})$.

In the following, we use a K-valued register implementation to demonstrate how our linearizability checking approach works.

**Example 1 (K-Valued Register [5, Section 10.2.1]).** A $K$-valued ($K > 2$) single-writer single-reader register $R$ can be simulated using an array $B$ of $K$ binary single-writer single-reader registers. The possible values of $R$ are $\{O, 1, \ldots, K-1\}$. The value $i$ is represented by a 1 in the $i$th entry of array $B$ and 0 in all other entries. For each binary register, there is a single processor (the writer) that can write to it and a single processor (the reader) that can read from it, and the values read or written can only be 0 or 1.

When read and write operations do not overlap, it is simple to perform the operations, i.e., a read operation scans the array beginning with index 0 until it finds a 1 in some entry and returns the index of this entry. A write operation writes the value $v$ by writing the value 1 in the entry whose index is $v$ and clearing (setting to 0) the entry corresponding to the previous value if different from $v$.

When read and write operations might overlap, to ensure that a read operation reads the last value written two changes are made: 1) a write operation clears only the entries whose indices are smaller than the value it is writing, and 2) a read operation does not return when it finds the first one but makes sure that all lower indexed bits are still zero. Specifically, the reader scans from the low indices toward the high indices until it finds the first one; then it reverses direction and scans back down to the beginning, keeping track of the smallest index observed to contain a 1 during the downward scan. This is the value returned. Details are given in Algorithm 3.2. Note that the linearization point of the $read$ operation is not fixed because the value of $v$ to be returned depends on the last position of value 1 found during the downwards scan, i.e., only the last execution of line 8 is the linearization point rather than any execution of line 8 is the linearization point.

The linearizable specification model is defined as in Algorithm 3.1, where $R$ is the shared register with initial value 0. The statement in Line 1 for both $read$ and $write$ operations is the linearization action.

The refinement relation is based on comparing the arguments of the invocation and return values of the responses of an operation. In this example, the only visible actions are the invocation and response of the read and write operations. The system model is constructed as a general client on the shared register, which consists of a parallel composition of a reader process and a writer process. Further, the writer process nondeterministically chooses one of $K$ possible values to execute write operation. If there are multiple operations a process can perform, then a process is modeled to nondeterministically execute one of the operations.

We remark that interesting progress properties can be verified by model checking. For example, suppose $K$ is 3, then one can model check the formula

$$\Phi inv(read)_{reader} \rightarrow \Psi(res(read, 0)_{reader} \vee res(read, 1)_{reader} \vee res(read, 2)_{reader}),$$

where $\Phi$ and $\Psi$ are modal operators denoting "always" and "eventually," respectively. This property says that once reader invokes the read operation, it will eventually get the value rather than being blocked by the writer indefinitely.

**Algorithm 3.1** K-valued register specification

```
shared $R := 0$;
Procedure read                 Procedure write($v$)
1: $v := R$;                   1: $R := v$;
2: return $v$                  2: return
```

$Reader := read; Reader$
$Writer := (write(0) \; [] \cdots [] \; write(K-1)); Writer$[8]
$System := Reader \| \; Writer$[9]

7. CAS is an atomic CPU instruction used in multithreading to achieve synchronization. It compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of that memory location to a given new value. The atomicity guarantees that the new value is calculated based on up-to-date information; if the value had been updated by another thread in the meantime, the write would fail.

8. [] denotes nondeterministic choice between operations.

9. $\|$ denotes the parallel composition of processes.

**Algorithm 3.2** K-valued register implementation
Initially the shared registers $B[0]$ through $B[K-1]$ are all 0.

```
Procedure read
1: $i := 0$;
2: while $B[i] = 0$ do
3:     $i := i + 1$;
4: end while
5: $up, v := i$;
6: for $i = up - 1$ downto 0 do
7:     if $B[i] = 1$ then
8:         $v := i$;
9:     end if
10: end for
11: return $v$

Procedure write($v$)
1: $B[v] := 1$;
2: for $i := v - 1$ downto 0 do
3:     $B[i] := 0$;
4: end for
5: return

$Reader := read; Reader$
$Writer := (write(0) \, [] \ldots [] \, write(K-1)); Writer$
$System := Reader \mid\mid Writer$
```

### 3.2.3 Linearizability Formalization

The following theorem characterizes the linearizability of an implementation $\mathcal{M}_{im}$ through a refinement relation where the corresponding specification $\mathcal{M}_{sp}$ is created by following the steps in Section 3.2.1 and $L_{im}$ and $L_{sp}$ are the LTSs corresponding to $\mathcal{M}_{im}$ and $\mathcal{M}_{sp}$. This theorem establishes our approach to verifying linearizability. Different versions of this result have appeared in distributed computing literature, e.g., Theorems 13.3, 13.4, and 13.5 in Lynch's book [40].

Here, we assume that all shared objects have finite domains and each process has finitely many local states, which disallows unbounded nontail recursion that results in infinitely many local states. These assumptions ensure that the constructed LTSs of the models have a finite number of states so that the models can be verified using model checking techniques. We further assume that there is no process creation or termination, which ensures that optimization techniques presented in Section 4 are applicable. This assumption is true for all the algorithms in this paper and generally holds when each process is designed to run on a single thread.

**Theorem 1.** *Let $L_{im}$ be an implementation LTS generated by the steps in Section 3.2.2, and $L_{sp}$ be the corresponding specification LTS generated by the steps in Section 3.2.1. All traces of $L_{im}$ are linearizable if and only if $L_{im} \sqsupseteq_T L_{sp}$.*

**Proof (Sufficient Condition).** For any trace $\sigma \in traces(L_{im})$, because $L_{im} \sqsupseteq_T L_{sp}$, $\sigma$ is also a trace of $L_{sp}$. Let $\rho$ be an execution history of $L_{sp}$ that generates the trace $\sigma$. We define the sequential permutation $\pi$ of $\sigma$ as the reordering of operations in $\sigma$ by following the order of the linearization actions of all operations and all processes in $\rho$. That is, if $op_1 <_\sigma op_2$, the linearization action of $op_1$ must be ordered before the linearization action of $op_2$ in $\rho$, and thus $op_1 <_\pi op_2$. It is also easy to verify that $\pi$ is a legal sequential history of object $o$ since the linearization action of every operation in $\rho$ is the only action in the operation that affects the object state based on its sequential specification and the order of operations in $\pi$ respects the order of linearization actions in $\rho$.

*Necessary condition.* Let $\sigma$ be a trace of $L_{im}$. By assumption, $\sigma$ is linearizable. We need to show that $\sigma$ is also a trace of $L_{sp}$. Since $\sigma$ is linearizable, there is a sequential permutation $\pi$ of $\sigma$ such that $\pi$ respects both the sequential specification of object $o$ and the runtime ordering of the operations in $\sigma$. We construct an execution history $\rho$ of $L_{sp}$ from $\sigma$ and $\pi$ as follows: Starting from the first action of $\sigma$, for any action $e$ in $\sigma$, 1) if it is an invocation action, append it to $\rho$; 2) if it is a response action $res(op, resp)_i$, locate the operation $op$ in $\pi$ and, for each unprocessed operation $op'$ by a process $j$ before $op$ in $\pi$, process $op'$ by appending a linearization action $lin(op')_j$ to $\rho$, following the order of $\pi$; finally, append $lin(op)_i$ and $res(op, resp)_i$ to $\rho$. It is not difficult to show that the execution history $\rho$ constructed this way is indeed a history of $L_{sp}$. Moreover, obviously the trace of $\rho$ is $\sigma$. Therefore, $\sigma$ 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. This provides the theoretical foundation of our method. Notice that the verification by refinement given above does not require identifying low-level actions in the implementation as linearization points, which can be difficult (or even impossible) for some algorithms (e.g., the elimination backoff stack [29], the restricted double-compare single-wrap operation [28]). In fact, the verification can be automatically carried out without special knowledge about the implementation beyond the implementation code itself.

## 3.3 Linearizability with Linearization Points

In some cases, one may be able to identify certain actions in an implementation as linearization points, which are *linearization actions*. This section presents an alternative and simpler way of formalization when the linearization points are known.

### 3.3.1 Linearizable Specification

When the linearization points are known, a linearizable specification can be constructed in a similar way as in Section 3.2.1. The difference is that we make linearization actions visible and hide the invocation and response actions. More specifically, we obtain a specification LTS $L'_{sp}$ by the following two modifications to $L_{sp}$: 1) We change each linearization action $lin(op)_i$ to $lin(op, resp)_i$ to include the response $resp$ computed by this linearization action; this is possible because the return value of an operation is available after linearization action; and 2) we make only linearization actions visible and all $inv(op)_i$ and $res(op, resp)_i$ invisible.

### 3.3.2 Implementation Formalization

To formalize any given concurrent algorithms in this case, we adopt the same way as for specification construction. After generating the $L_{im}$ by following Section 3.2.2, we mark only linearization actions visible and hide all other actions as above. There is no need to add invocation and response actions in this case. Similarly to the construction of linearizable specifications, we include the responses in the linearization actions. We demonstrate this idea using the following example.

**Example 2 (Abstract Concurrent Counter).** Treiber [53] proposed a concurrent stack implementation using CAS instructions. Here, we use one of its simplified versions presented in [10], as shown in Algorithm 3.3. The pointer $H$ is shared by all processes. Each operation tries to update $H$ until its CAS operation succeeds. To keep this algorithm finite-state (hence subject to model checking), we assume that the size of the stack and the number of processes are finite. Based on this algorithm, we propose a concurrent counter implementation by abstracting the stack elements to stack size as shown in Algorithm 3.4. A proof that Algorithm 3.4 is equivalent to the original algorithm, Algorithm 3.3, is omitted for brevity.

Since the concurrent counter implements the standard *push* and *pop* operations, the specification of the concurrent counter algorithm is defined in Algorithm 3.5. Here, the actual data are abstracted because only the stack size is relevant, which is modeled as shared variable $S$ with initial value 0. A *push* operation increases the stack size by 1. A *pop* operation decreases the stack size by 1 if it is bigger than 0 and returns the stack size before the decrease; otherwise it returns 0. We introduce *atomic* construct to indicate that the program in the block is to be executed as one superstep, noninterleaved with other processes. This atomic construct is the linearization action for the *pop* operation.

The linearization points of Algorithm 3.3 are known [3]. Therefore the verification can be conducted directly by modeling linearization points and leaving out invocation and response actions. Clearly the specification model of the stack has only one linearization action (i.e., the corresponding atomic block) for each operation. We make each linearization action visible and include the return value. For the *push* operation, its linearization action is $lin(push, S+1)_i$, where $i$ is the process identifier and $S+1$ is the stack size after update. Likewise, the linearization action of the *pop* operation is $lin(pop, S)_i$, where $i$ is also the process identifier and $S$ is the stack size before update.

The linearization points of the counter implementation are conditional. For the *push* operation, only a successful CAS (at line 4) is considered to be a linearization point. For the *pop* operation, there are two conditional linearization points: If the counter is 0, returning 0 at line 4 is a linearization point; otherwise, a successful CAS (line 7) is a linearization point.

**Algorithm 3.3** Concurrent stack implementation

```
type $Node = \{val : T; next : Node\}$;
shared $Node H := null$;
N is the maximum value that can be stored by the stack
Procedure push($v$)
1: $n := newNode()$;
2: $n.val := v$;
3: repeat
4:     $ss := H$;
5:     $n.next := ss$;
6: until $CAS(H, ss, n)$
7: return

Procedure pop
1: repeat
2:     $ss := H$;
3:     if $ss = null$ then
4:         return $empty$
5:     end if
6:     $n := ss.next$;
7:     $lv := ss.val$;
8: until $CAS(H, ss, n)$
9: return $lv$

$Process(i) := (push(1) [] \ldots [] push(N) [] pop); Process(i)$
$System := Process(1) || Process(2) || \ldots || Process(N)$
```

**Algorithm 3.4** Concurrent counter implementation

```
shared $H := 0$;
Procedure push
1: repeat
2:     $ss := H$;
3:     $n := ss + 1$;
4: until $CAS(H, ss, n)$
5: return

Procedure pop
1: repeat
2:     $ss := H$;
3:     if $ss = 0$ then
4:         return 0;
5:     end if
6:     $n := ss - 1$;
7: until $CAS(H, ss, n)$
8: return $ss$;

$Process(i) := (push [] pop); Process(i)$
$System := Process(1) || Process(2) || \ldots || Process(N)$
```

**Algorithm 3.5** Concurrent counter specification

```
shared $S := 0$;
Procedure push
1: $S := S + 1$;
2: return

Procedure pop
1: atomic
2:     if $S == 0$ then
3:         $v := 0$;
4:     else
5:         $S := S - 1$;
6:         $v := S$;
7:     end if
8: end atomic
9: return $v$;

$Process(i) := (push [] pop); Process(i)$
$System := Process(1) || Process(2) || \ldots || Process(N)$
```

### 3.3.3 Linearizability Formalization

**Theorem 2.** *Let $L'_{im}$ be an implementation LTS with known linearization actions and specified by the steps in Section 3.3.2, and $L'_{sp}$ be the corresponding specification LTS generated by the steps in Section 3.3.1. All traces of $L'_{im}$ are linearizable if and only if $L'_{im} \sqsupseteq_T L'_{sp}$.*

**Proof (Sufficient condition).** For any trace $\sigma \in traces(L'_{im})$, because $L'_{im} \sqsupseteq_T L'_{sp}$, $\sigma$ is also a trace of $L'_{sp}$. Let $\rho$ be the execution history of $L'_{im}$ that generates the trace $\sigma$. $\sigma$ respects the sequential specification of the objects since the linearization action in $L'_{sp}$ is the only action that affects the object states. Furthermore, each linearization action in $\sigma$ is always between its invocation and response action in $\rho$ because of the way $\sigma$ is generated. According to the second definition of linearizability using linearization points in Section 2, $\sigma$ is the shrank execution of $\rho$, and each action is the linearization point of the corresponding operation.

*Necessary condition.* Let $\sigma$ be a trace of $L'_{im}$. Let $\rho$ be the execution history of $L'_{im}$ that generates the trace $\sigma$. By assumption, $L'_{im}$ is linearizable and all linearization actions are identified in the implementation. Therefore, $\sigma$ is the shrank execution of $\rho$ and each action is the linearization point of the corresponding operation. Condition "all linearization actions are identified in the implementation" ensures that each operation has a linearization point in $\sigma$. Since linearization actions in $L'_{im}$ represent the effects of the object changes, we can find the same linearization action in the $L'_{sp}$. Hence, $\sigma$ is also a trace of $L'_{sp}$ □.

It is not difficult to see that the implementation model built with the knowledge of linearization points is much simpler and contains fewer visible actions. Hence, the verification can be done more efficiently by comparing only one action for each operation. However, it is important to note that, as stated in Theorem 2, 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 determined linearization points have to be complete. Otherwise, even if the verification finds a counterexample for the refinement relation, it may be due to unidentified linearization points, and one cannot conclude that the implementation is not linearizable.

## 4 Verification of Linearizability

With the results presented in Section 3, all we need is a scalable refinement checking algorithm to establish linearizability. In this section, we present a classic algorithm [45] for refinement checking, and then optimize it with symmetry reduction, partial order reduction, and their combination.

In the following, we fix two LTSs, $L_{im} = (S_{im}, init_{im}, Act_{im}, T_{im})$ and $L_{sp} = (S_{sp}, init_{sp}, Act_{sp}, T_{sp})$, which represent an implementation and a specification, respectively. Notice that both $L_{im}$ and $L_{sp}$ typically have invisible actions. That is, if linearization points are unknown, all actions except invocation and response actions are invisible; if linearization points are known, all except the linearization actions are invisible. As a result, both $L_{im}$ and $L_{sp}$ have a degree of nondeterminism, e.g., two different processes both can take $\tau$-transitions, resulting in two identically labeled actions from the same state.

For ease of understanding, we use the bounded abstract concurrent counter algorithm as a running example, and show how the state space of the whole program can be reduced by symmetry reduction, then partial order reduction, and, at last, their combination. Moveover, to show the generality of our approach, we assume that the linearization points are unknown. To properly display the entire state space, due to space limitations, we only show a stack being used by two processes, with ids 0 and 1, respectively, so the effect of reduction approaches can be demonstrated visually to help readers better understand the technical details of our approach. Furthermore, we require that each process perform only one *push* operation. The stack is initially empty and its size is 2.

### 4.1 A Linearizability Checking Algorithm

To establish a refinement relationship between $L_{im}$ and $L_{sp}$, we need to show that every trace of $L_{im}$ is allowed by $L_{sp}$. Because of nondeterminism in $L_{sp}$, after a sequence of visible actions there may be many states that the system might be in. A refinement checking algorithm thus will have to keep track of all the states reachable in $L_{sp}$ on a given trace, which can be achieved by determinization, also known as normalization. A determinization of an LTS $L$ is a deterministic LTS, written as $D(L)$, such that $L$ and $D(L)$ have the same traces. With determinization, checking whether $L_{im}$ refines $L_{sp}$ is reduced to checking whether $L_{im}$ refines $D(L_{sp})$, which is easier because there is exactly one state in $D(L_{sp})$ corresponding to each possible trace. A standard approach for determinization is through subset construction [32].

**Definition 5 (Determinization).** *Let $L = (S, init, Act, \rightarrow)$ be an LTS. The determinized LTS of $L$ is $D(L) = (S_d, init_d, Act_d, \rightarrow_d)$ where $S_d \subseteq 2^S$ is a set of subsets of $S$, $init_d = \tau^*(init)$, $Act_d = vis_{D(L)} = vis_L$, and $\rightarrow_d \subseteq S_d \times Act_d \times S_d$ is a transition relation such that $X \overset{\alpha}{\rightarrow}_d Y$ iff $Y = \{y : S | \exists x \in X, \exists s \in S : x \overset{\alpha}{\rightarrow} s \wedge y \in \tau^*(s)\}$.*

In the following, we fix $D_{sp} = (S_d, init_d, Act_d, T_d)$ to be the determinized LTS of $L_{sp}$. All states connected by $\tau$-transitions in $L_{sp}$ are grouped in $D_{sp}$. For instance, the dotted circles in Fig. 1 show the determinized states. It is straightforward to show that $D_{sp}$ is deterministic, i.e., for any state $s \in S_d$ and any visible action $\alpha \in Act_d$, there is at most one state $s' \in S_d$ such that $s \overset{\alpha}{\rightarrow}_d s'$.

Refinement checking is then reduced to reachability analysis[10] of the synchronous product of $L_{im}$ and $D_{sp}$. Each state of the product space is a state pair $(im, sp)$, where $im$ is an implementation state and $sp$ is a determinized specification state. If there exists a state pair $(im', sp')$ such that $im \overset{\alpha}{\rightarrow}_{im} im'$, $sp \overset{\alpha}{\rightarrow}_d sp'$ for some $\alpha \in Act_d \cap Act_{im}$, we say there is a *product transition* from $(im, sp)$ to $(im', sp')$ labeled with $\alpha$.

Algorithm 4.1 shows an on-the-fly linearizability checking algorithm. It performs a depth-first-search for a state that violates linearizability, i.e., a pair $(im, sp) \in S_{im} \times S_d$ such that $sp$ is an empty set. The algorithm returns true if no such pair is found. Otherwise, a counterexample violating trace refinement is found. The algorithm maintains two data structures. *checked* is a set of product states that have been explored and *pending* is a stack containing new states yet to be explored. On line 2, the initial state of the product is pushed into *pending*. While there are new states to be explored (i.e., *pending* is not empty), a pending state is obtained from *pending* on line 7, which is then added to *checked* on line 8. If the state is of the form $(im, \emptyset)$, we infer that there exists a trace that leads $L_{im}$ to state $s$ and leads $L_{sp}$ to no state, and therefore the trace serves as a counterexample to refinement. The code fragments for producing a counterexample are shown in gray color, which stores a path of $L_{im}$ from the initial state $init_{im}$ to state $s$ in stack *counterexample*. For brevity, these code fragments are omitted in the algorithms presented later in this section. If a successor state of $(im, sp)$ has not been explored (i.e., $(im', sp') \notin checked$ on line 20), it is pushed into *pending* on line 21. Note that Function $next(im, sp)$ returns the set of successor states in the product. Formally,

10. Reachability analysis is to check whether a particular state can be reached from the initial state of an LTS via every execution trace.

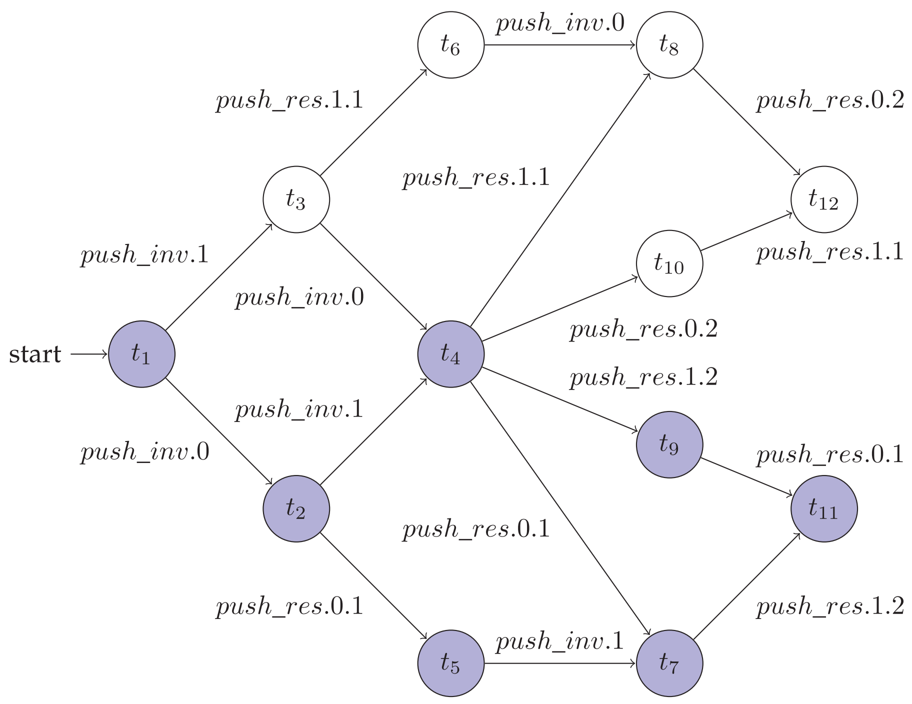

Fig. 2. Determinization of the specification LTS.

$$\begin{aligned} &next(im, sp)\\ &\quad = \{(im', sp) | im \overset{\alpha}{\rightarrow}_{im} im' \wedge \alpha \in invis_{L_{im}}\}\\ &\qquad \cup \{(im', sp') | \exists \alpha \in vis_{L_{im}} : im \overset{\alpha}{\rightarrow}_{im} im'\\ &\qquad \wedge \forall x' \in sp', \exists x \in sp, \exists u \in sp' : x \overset{\alpha}{\rightarrow}_{sp} u \wedge x' \in \tau^*(u)\}. \end{aligned}$$

If the implementation takes a $\tau$-transition (e.g., any action that is not an invocation or response action if the linearization points are unknown), the trace of the implementation remains the same (as $\tau$ is always pruned from the trace) and therefore the set of corresponding states in $L_{sp}$ (which are reached via the same trace) remains the same. In other words, the determinized state remains the same. If the implementation takes a visible transition, then the same action must be performed by the specification, resulting in a new determinized state. To obtain $next(s, X)$ algorithmically, it is necessary to compute the set of states reached by a $\tau$-transition from a given state, which can be implemented using a standard depth-first-search.

**Algorithm 4.1.** A linearizability checking algorithm

```
**Procedure Linearizability** $(L_{im}, L_{sp})$
1:  $checked := \emptyset$
2:  $pending.push((init_{im}, \tau^*(init_{sp})))$
3:  $stack\_depth.push(0)$
4:  $path\_depth := \langle\rangle$
5:  $counterexample := \langle\rangle$
6:  while $pending \neq \emptyset$ do
7:      $(im, sp) := pending.pop()$
8:      $checked := checked \cup \{(im, sp)\}$;
9:      $d := stack\_depth.pop()$
10:     while $d > 0 \wedge path\_depth.peek() \geq d$ do
11:         $path\_depth.pop()$
12:         $counterexample.pop()$
13:     end while
14:     $path\_depth.push(d)$
15:     $counterexample.push(im)$
16:     if $sp = \emptyset$ then
17:         return $counterexample$
18:     end if
19:     for all $(im', sp') \in next(im, sp)$ do
20:         if $(im', sp') \notin checked$ then
21:             $pending.push((im', sp'))$
22:             $stack\_depth.push(d + 1)$
23:         end if
24:     end for
25: end while
26: return $true$
```

For the running example, Fig. 2 shows the determinization of its specification LTS and Fig. 3 shows the specification-implementation product LTS explored during linearizability checking (let us ignore for now the colors and shades of edges and nodes). Each node in Fig. 3 contains two parts, the implementation state in the upper part and the determinized specification state in the lower part. If an action is a visible action, then the corresponding edge is labeled with its name. Otherwise, the edge is labeled with the statement in the *push* operation of Algorithm 3.4. $l_i^j$ denotes the statement on line $i$ executed by process $j$. Because we assume the linearization points of this algorithm are unknown, the visible actions are the invocation and response of the *push* operation. That is, $push\_inv.i$ and $push\_res.i.v$ are the invocation and response action, respectively, where $i$ is the identifier of the invoking process and $v$ is the return value of the *push* operation.

The algorithm terminates as long as the product has finitely many states. The soundness of the algorithm follows from [45]. Note that determinization is performed on-the-fly so that in the presence of a counterexample only part of $D_{sp}$ is constructed. In practice, this algorithm may suffer from state space explosion. Its complexity is linear in the number

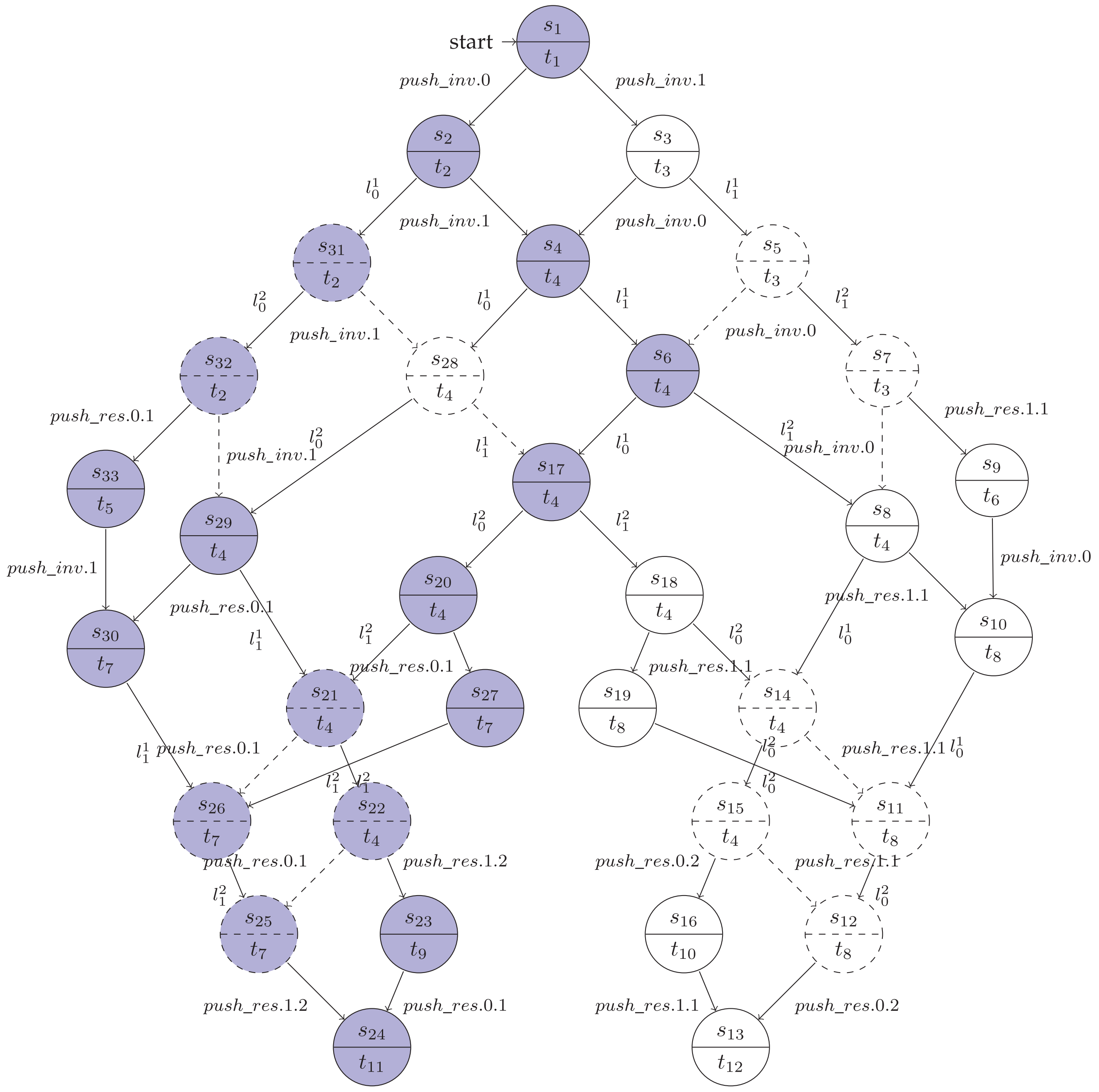

Fig. 3. Specification-implementation product LTS of concurrent stack implementation.

of transitions in the product. The size of the product is bounded by the size of $L_{im}$ and $D_{sp}$. In the worst case, $D_{sp}$ may have exponentially more states than $L_{sp}$. Further, $L_{im}$ and $L_{sp}$ typically have an exponential blow up in the size of the system models representing the specification and implementation. Thus, it is necessary to explore powerful state reduction techniques to model check complex concurrent object implementations.

### 4.2 Optimization 1: Symmetry Reduction

A concurrent data object is often designed to be accessed by many behaviorally similar or even identical processes. Such similarity or symmetry often induces equivalent portions of the underlying state space. Symmetry reduction [24] is an effective technique for eliminating such equivalent states. The idea is to only explore the behavior of one process and conclude the same for homogeneous others (subject to property-specific conditions).

For temporal logic model checking, classical symmetry reduction approaches [34], [48], [23] often choose a unique representative state from each class of equivalent states. Each visited state is replaced with its representative state. Unfortunately, these approaches fail to fit into the context of refinement-style linearizability checking. The significant obstacle is lock-step synchronization on all visible actions between the state spaces of implementation and specification models. If symmetry reduction is applied to an implementation (respectively, a specification) model, replacing an implementation (respectively, specification) state with its representative potentially influences the synchronization result and thus sacrifices the validity of linearizability checking. For instance, suppose a common action $\alpha$ is ready to be executed from a state pair $(s, s')$, where $s$ is an implementation state and $s'$ is a specification state. If symmetry reduction is applied to the implementation model, the state pair is changed to $(t, s')$, where $t$ is the representative state of the equivalence class including $s$. However, $t$ is likely unable to engage $\alpha$. In this case, a false counterexample is produced by Algorithm 4.1.

Moffat et al. [43] proposed a symmetry reduction approach for trace refinement checking. In the following, we first use their approach to perform symmetry reduction on $L_{im}$ during linearizability checking. Then, we extend and improve their work to achieve better performance.

We begin with introducing relevant notations and terminology. A permutation $\gamma$ on a finite set of objects (e.g., process identifiers) is a bijection from the set to itself (i.e., a function that is one-to-one and onto). For instance, suppose the system is a parallel composition of $N$ processes $P_i$, where $i$ ranging from 1 to $N$ is a unique identifier associated with the process. Permutation $\gamma$ may be defined such that $\gamma(i) = (i+1) \bmod N$ for each $i$. Suppose $N$ is 2. We have $\gamma(1) = 2$ and $\gamma(2) = 1$, written as $(1, 2)$.[11] A permutation group is a group of permutations under functional composition $\circ$. For instance, a permutation group in the above example formed by $\gamma$ is $\langle (1,2), (1,2) \circ (1,2) \rangle$, which equals $\langle (1,2), (1)(2) \rangle$.

Let $Perm(S)$ be the group of permutations of the finite set $S$. Next, we define the concept of automorphism group.

**Definition 6 (Automorphism).** *Given an LTS $L = (S, init, Act, \rightarrow)$, a group $G \leq Perm(s) \times Perm(Act)$ is an automorphism group of $L$ if and only if*

1. $$\forall \gamma \in G, s_1, s_2 \in S, \alpha \in Act :$$
$$s_1 \xrightarrow{\alpha} s_2 \Longrightarrow \gamma(s_1) \xrightarrow{\gamma(\alpha)} \gamma(s_2).$$

2. $\gamma(init) = init.$

It is often the case that an automorphism group is given as a group acting on the process identifiers of the state variables as well as labeled actions. For example, a permutation $\gamma$, defined on the actions and states in Fig. 2, may also be described as the permutation of process identifiers, i.e., $(0, 1)$.

Our starting point of symmetry reduction is the simple observation that, in practice, each operation on a concurrent data object often originates from a generic system description without discriminating process identifiers in both its implementation and specification. This means that if there is any symmetry relation between processes in an implementation model, then there will be the same relation in its corresponding specification model and vice versa. Note that both the implementation and specification models in this work take the form of a parallel composition of processes. The insight is therefore captured by a premise: *A permutation is an automorphism of $L_{im}$ if and only if it is also an automorphism of $L_{sp}$.*

11. Cycle notation is used to write down a permutation such that the contents in parentheses move in a cycle. For example, $(1, 2)$ denotes moving 1 to 2, and 2 back to 1.

Let $G$ be an automorphism group on both $L_{im}$ and $L_{sp}$. Considering that the usual linearizability checking algorithm explores the product space of $L_{im}$ and $D_{sp}$, we need to prove that $G$ is also an automorphism group on this product space so that we can apply symmetry reduction on it. We first prove the following lemma.

**Lemma 1.** *If a permutation is an automorphism of an LTS $L$, it is an automorphism of $D(L)$.*

**Proof.** Suppose that permutation $\gamma$ is an automorphism of $L$. By definition, a transition $t = X \xrightarrow{\alpha}_d Y$ is in $D(L)$ iff $\exists x \in X, \exists y \in Y : x \xrightarrow{\alpha} x' \wedge y \in \tau^*(x')$. Further, $\forall s_1, s_2 \in S$, $\alpha \in Act : s_1 \xrightarrow{\alpha} s_2$ iff

$$\gamma(s_1) \xrightarrow{\gamma(\alpha)} \gamma(s_2).$$

Thus, $x \xrightarrow{\alpha} x' \Leftrightarrow \gamma(x) \xrightarrow{\gamma(\alpha)} \gamma(x')$ and $y \in \tau^*(x') \Leftrightarrow \gamma(y) \in \tau^*(\gamma(x'))$. Thus, there is exactly a transition

$$\gamma(P) \xrightarrow{\gamma(\alpha)}_d \gamma(Q)$$

in $D(L)$. Considering that $t$ is arbitrary, $\gamma$ is also an automorphism of $D(L)$. □

Next, we prove the following theorem:

**Theorem 3.** *If $G$ is an automorphism group on both $L_{im}$ and $L_{sp}$, then it is an automorphism group on the product space of $L_{im}$ and $D_{sp}$.*

**Proof.** By Definition 6, we must show that for any $\gamma \in G$, 1) if $sp \xrightarrow{\alpha}_d sp'$ and $im \xrightarrow{\alpha}_{im} im'$, then $\gamma(sp) \underset{d}{\xrightarrow{\gamma(\alpha)}} \gamma(sp')$ and $\gamma(im) \xrightarrow{\gamma(\alpha)}_{im} \gamma(im')$, 2) $\gamma(init_d) = init_d$ and $\gamma(init_{sp}) = init_{sp}$.

By the premise, $\gamma$ is an automorphism of $L_{im}$. If $im \xrightarrow{\alpha}_{im} im'$, then $\gamma(im) \xrightarrow{\gamma(\alpha)}_{im} \gamma(im')$. By Lemma 1, $\gamma$ is an automorphism of $D_{sp}$. So if $sp \xrightarrow{\alpha}_d sp'$, then $\gamma(sp) \xrightarrow{\gamma(\alpha)}_d \gamma(sp')$. Therefore, $\frac{\gamma(sp)\gamma(\alpha)}{\rightarrow_d \gamma(sp')}$ and $\gamma(im) \xrightarrow{\gamma(\alpha)}_{im} \gamma(im')$.

Similarly, because $\gamma$ is an automorphism of both $L_{im}$ and $D_{sp}$, $\gamma(init_d) = init_d$ and $\gamma(init_{sp}) = init_{sp}$.

Since $\gamma$ is an arbitrary automorphism in $G$, we conclude that $G$ is an automorphism group on the product space of $L_{im}$ and $D_{sp}$. □

The function *repPair* is defined to "twist" a state pair in the product space using an automorphism in $G$ as follows:

$$repPair(im, sp, \gamma) = (\gamma(im), \gamma(sp))$$
$$\text{where } im \in L_{im} sp \in D_{sp} \text{ and } \gamma \in G.$$

A *repPair-twisted* (hereafter *twisted*) path through the product space is a sequence $\langle s_0, \alpha_1, \gamma_1, s_1, \ldots, s_{n-1}, \alpha_n, \gamma_n, s_n \rangle$ of states, actions, and permutations, starting and ending with states such that: For all $0 \leq i < n$, suppose $s_i = (im, sp)$, there exists a state pair $(im', sp')$ and $\gamma_{i+1} \in G$ such that $im \xrightarrow{\alpha_{i+1}}_{im} im', sp \xrightarrow{\alpha_{i+1}}_d sp'$ and $s_{i+1} = (\gamma_{i+1}(im'), \gamma_{i+1}(sp')) = repPair(im', sp', \gamma_{i+1})$. In this way, function *repPair* "twists" the original path explored in the usual refinement checking. For example, there exists a twisted path $\langle (t_1, s_1), push\_inv.0, (0, 1), (t_3, s_3), \tau, (0, 1), (t_2, s_{31}), push\_inv.1, (0, 1), (t_4, s_6) \rangle$ in Fig. 3.

To generate a representative state for each visiting implementation state, we assume the existence of a function $rep$ such that, for each $s \in S$, $rep(s)$ is a representative of $s$. To perform symmetry reduction on $L_{im}$ during linearizability checking, we restrict the definition of $repPair$ using $rep$ as follows: $repPair(im, sp, \gamma) = (rep(im), \gamma(sp))$ such that $\gamma(im) = rep(im)$.

Algorithm 4.2 is developed to perform on-the-fly linearizability checking with symmetry reduction. The underlined text shows the differences compared with Algorithm 4.1. Function $Rep1$ calculates a unique representative of each visited state, i.e., $Rep1(s) = (s', \gamma)$ such that $\gamma(s) = s'$. Each $\gamma$ used in $repPair$ can be stored through the explored path to recover a twisted counterexample path to an actual path in $L_{im}$. Assume a linearizability violated pair $s = (im, \emptyset)$ is found and $\pi$ is the current explored twisted path arriving at $s$. $\pi$ can be easily composed by concatenating all elements popped off the stack $pending$ in reverse order. Let $\pi$ be $\langle s_0, \alpha_1, \gamma_1, s_1, \ldots, \alpha_n, \gamma_n, s_n\rangle$, then function $recover$ is used to "recover" it to an actual counterexample path in the original product space. Formally, the function $recover$ is defined as follows:

$$recover(\pi) = \langle s_0, \alpha_1, \gamma_1^{-1}s_1, \ldots, (\gamma_1^{-1}\gamma_2^{-1}\cdots\gamma_{n-1}^{-1})\alpha_n, (\gamma_1^{-1}\gamma_2^{-1}\cdots\gamma_n^{-1})s_n\rangle.$$

The following theorem (which is based on Lemma 2) guarantees the soundness and completeness of Algorithm 4.2. Algorithm 4.2 finds a twisted counterexample path exactly when the refinement does not hold.

**Lemma 2 [43].** *Suppose function repPair maps each state pair $(u, v)$ to $(\gamma(u), \gamma(v))$ for some $\gamma \in G$. Then, for all paths $\pi$, there is a path to state pair $s = (im, sp)$ if and only if there is a repPair-twisted path $\pi'$ to state pair $\gamma(s) = (\gamma(im), \gamma(sp))$, with $recover(\pi') = \gamma(\pi)$, for some $\gamma \in G$.*

**Theorem 4 [43].** *Suppose function repPair maps each state pair $(u, v)$ to $(\gamma(u), \gamma(v))$ for some $\gamma \in G$. $L_{im} \sqsupseteq_T D_{sp}$ has a counterexample path $\pi$ if and only if it has a counterexample repPair-twisted path $\pi'$ with $recover(\pi') = \pi$.*

Besides the above symmetry reduction, we observe that in practice for a state pair $(im, sp)$, there may exist multiple permutations $\gamma_1, \gamma_2, \ldots, \gamma_n \in G$ such that for every $i$ in $\{1, \ldots, n\} : \gamma_i(s) = rep(im)$, and there exists $i, j : i \neq j$, $\gamma_i(s) \neq \gamma_j(s)$. That is, different permutations may produce different twisted pairs with the same representative implementation state for a state pair in Algorithm 4.2. To check whether counterexample searching is sensitive to the permutations chosen, we develop the following theorem.

**Theorem 5.** *For any state $s = (im, sp)$ in the product space of $L_{im}$ and $D_{sp}$, if there exist $\gamma_1, \gamma_2 \in G$ and $\gamma_1 \neq \gamma_2$, then there is a twisted path from $\gamma_1(s)$ to $q$ if and only if there exists a twisted path from $\gamma_2(s)$ to $\gamma(q)$, for some $\gamma \in G$.*

**Proof.** We will prove that there is a *twisted* path $\pi = \langle s_0 = \gamma_1(s), \alpha_1, \sigma_1, \ldots, \alpha_n, \sigma_n, s_n\rangle$ if and only if there exists a *twisted* path $\pi' = \langle s'_0 = \gamma_2(s), \alpha'_1, \sigma'_1, \ldots, \alpha'_n, \sigma'_n, s'_n\rangle$ such that $q = s_n$, and for every $0 \leq i \leq n$, there exists $\gamma \in G : \gamma(s_i) = s'_i$.

*Necessary condition.* Induction on $|\pi|$.

*Basis.* $|\pi| = 0$. Then, $\pi = \langle\gamma_1(s)\rangle$. There is exactly one *twisted* path $\pi' = \langle\gamma_2(s)\rangle$. Thus, there exists $\gamma = \gamma_1^{-1}\gamma_2 \in G : \gamma_1\gamma(s) = \gamma_2(s)$.

*Induction hypothesis.* Assume that the claim is true for any *twisted* path $\pi$ such that $|\pi| \leq k$.

*Induction step.* We show it also holds for all *twisted* paths $\pi$ where $|\pi| = k + 1$.

Consider a *twisted* path $\pi$ of the form $\langle s_0 = \gamma_1(s), \alpha_1, \sigma_1, \ldots, \alpha_{k+1}, \sigma_{k+1}, s_{k+1}\rangle$. Then, there is a transition from $s_k$ to $pre\text{-}s_{k+1}$ labeled $\alpha$ where $s_{k+1} = \gamma_{k+1}(pre\text{-}s_{k+1})$. From the induction hypothesis, there is a *twisted* path $\pi' = \langle s'_0 = \gamma_2(s), \alpha'_1, \sigma'_1, \ldots, \alpha'_k, \sigma'_k, s'_k\rangle, \sigma'_{k+1}, s'_{k+1}$ such that $\gamma(s_k) = s'_k$. By Theorem 3, there is a transition from $\gamma(s_k)$ to $\gamma(pre\text{-}s_{k+1})$ labeled with $\gamma(\alpha)$. So, $s'_{k+1} = \gamma'_{k+1}\gamma(pre\text{-}s_{k+1}) = \gamma'_{k+1}\gamma\gamma_{k+1}^{-1}(s_{k+1})$.

*Sufficient condition.* Similar. □

An immediate corollary of Theorem 5 is shown as follows, which is the foundation of our improvement.

**Corollary 1.** *For any state $s = (im, sp)$ in the product space of $L_{im}$ and $D_{sp}$, if there exist $\gamma_1, \gamma_2 \in G$ such that $\gamma_1(s) = \gamma_2(s)$ and $\gamma_1 \neq \gamma_2$, then there exists a counterexample twisted path if and only if there exists a twisted path.*

Based on the above results, we develop a new Algorithm 4.3 to allow more state reduction. The underlined text shows the differences compared with Algorithm 4.2. Function $Rep2$ is defined as: $Rep2(s) = (s', \langle\gamma_1, \ldots, \gamma_n\rangle)$ such that for every $i \in \{1, \ldots, n\} : \gamma_i(s) = s'$. By Corollary 1, it is sufficient to explore only one of the states $(s', \gamma_i(sp))$. Thus, if none of these states have been explored on line 11, one state is pushed into $pending$ on line 12.

**Algorithm 4.2.** Linearizability checking algorithm with symmetry reduction

```
Procedure linearizability_sym1 $(L_{im}, L_{sp})$
1:  $checked := \emptyset$;
2:  $pending.push((init_{im}, \tau^*(init_{sp})))$;
3:  while $pending \neq \emptyset$ do
4:      $(im, sp) := pending.pop()$;
5:      $checked := checked \cup \{(im, sp)\}$;
6:      if $sp = \emptyset$ then
7:          return $false$
8:      end if
9:      for all$(im', sp') \in next(im, sp)$ do
10:         $\underline{(repIm', \gamma) = Rep1(im')}$
11:         if $\underline{(repIm', \gamma(sp'))} \notin checked$ then
12:             $pending.push(\underline{(repIm', \gamma(sp'))})$;
13:         end if
14:     end for
15: end while
16: return $true$
```

**Algorithm 4.3.** Improved Linearizability checking algorithm with symmetry reduction

```
Procedure linearizability_sym2 $(L_{im}, L_{sp})$
1:  $checked := \emptyset$;
2:  $pending.push((init_{im}, \tau^*(init_{sp})))$;
3:  while $pending \neq \emptyset$ do
4:      $(im, sp) := pending.pop()$;
```

```
5:     $checked := checked \cup \{(im, sp)\}$;
6:     **if** $sp = \emptyset$ **then**
7:         **return** $false$
8:     **end if**
9:     **for all**$(im', sp') \in next(im, sp)$ **do**
10:        $\underline{(repIm', \langle\gamma_0, \gamma_1, ..., \gamma_n\rangle) = Rep2(im')}$
11:        **if** $\underline{\forall 0 \le i \le n(repIm', \gamma_i(sp'))} \notin checked$ **then**
12:            $\overline{pending.push(}\underline{(repIm', \gamma_0(sp'))});$
13:        **end if**
14:    **end for**
15: **end while**
16: **return** $true$
```

In the following, we explain how Algorithm 4.3 works step by step on the concurrent stack example. It is not difficult to find that each pair of states in the symmetric positions are equivalent states in Figs. 2 and 3, i.e., one can be transformed to the other by process identifier permutation $(1, 2)$, e.g., $(1, 2)(t_3) = t_2$ . Further, we assume that each colored state in Fig. 3 is the representative state among the class of its equivalent states.

Starting from the initial state $(s_1, t_1)$, we first nondeterministically pick a transition to execute, the right one for example. At the resultant state $(s_3, t_3)$, we use function $repPair$ to twist it to get a new state to proceed. Because

$$rep(s_3) = s_2 = (1, 2)(s_3),$$
$$repPair(s_3, t_3) = (rep(s_3), (1, 2)(t_3)) = (s_2, t_2).$$

Starting from the product state $(s_2, t_2)$, we again nondeterministically pick one of these transitions, say the right one, and get to $(s_4, t_4)$. Because $rep(s_4) = s_4$, there is no need to change $t_4$. Again, if we pick the left transition from $(s_4, t_4)$ and arrive at $(s_{28}, t_4)$, we twist it to state $(s_6, t_4)$. By proceeding in this manner, we eventually explore all colored states in the product space so that we succeed in performing symmetry reduction on the implementation LTS and preserve the validity of linearizability checking.

### 4.3 Optimization 2: Partial Order Reduction

A process performs an atomic action at each step to move the system from one state to another. Partial order reduction is an effective state space reduction technique for concurrent systems with independent actions. The motivation is that the effect of independent concurrent actions is irrelevant to their interleaving orderings. If the property of interest does not depend on the intermediate states through the execution traces of these actions, a number of equivalent orderings of concurrent actions can be eliminated, which often yields a good reduction on the state space.

In practice, most concurrent object algorithms have a low degree of interprocess interaction and coordination for scalability reason. Many program statements are local computations or access disjoint locations of the shared data object. This loose coupling potentially induces many independent transitions and thus enables effective partial order reduction on these algorithms. In this section, we show how to perform partial order reduction for linearizability checking.

We adopt the recently proposed dynamic partial order reduction technique called Cartesian partial order reduction in [27] for linearizability checking for two reasons. First, pointer variables are used frequently in concurrent object algorithms. Static partial order reductions [35], [57], [26] fail to identify their independence precisely and thus cause a poor reduction on the state space. Second, concurrent algorithms with optimistic or lazy synchronization [30, chapter 9] put operation details within a loop. In the loop body, it tests synchronization conflict with other processes. If no conflict is found, the update will proceed; otherwise, it will go back to the start of the loop and retry. Dynamic partial order reduction [25] relies on a stateless search and thus cannot handle systems with loops, while a Cartesian one uses a stateful search and can handle loops.

For convenience, we describe the preceding notion of path in a more succinct notation by omitting immediate states, e.g., $\langle s_0, \alpha_1, \alpha_2, \ldots, \alpha_n, s_n\rangle$. A *legal path of process P* is a path that has at least one transition and all its transitions are executed by process $P$. Given an LTS $L = (S, init, Act, \rightarrow)$, for each $s$ in $S$ and $\alpha$ in $Act$, function $\alpha(s)$ returns the set of $\alpha$-successors of $s$. That is, $s' \in \alpha(s)$ iff $s \xrightarrow{\alpha} s'$.

The following defines the notion of actions being *independent*, which is central for any partial order reduction.

**Definition 7 (Independence).** *Given an LTS $L = (S, init, Act, \rightarrow)$ and $\alpha, \beta \in Act$ from different processes, $\alpha$ and $\beta$ are independent if for any $s \in S$ with $\alpha, \beta \in enabled(s)$: $\beta \in enabled(\alpha(s))$ , $\alpha \in enabled(\beta(s))$, and $\alpha(\beta(s)) = \beta(\alpha(s))$. $\alpha$ and $\beta$ are dependent if $\alpha$ and $\beta$ are not independent.*

For instance, in Algorithm 3.2, any pair of statements of concurrent $read$ operations is independent. Their ordering does not influence the execution result. In our setting, we define that two actions are dependent if two actions access the same variable, and at least one action writes the variable.

The standard semantics of a concurrent program can be regarded as controlled by a special scheduler. The scheduler nondeterministically picks one process to be executed after each transition. Cartesian semantics is proposed as a new operational semantics in cartesian partial order reduction [27] to bypass many unnecessary context switches and meanwhile to preserve soundness and completeness. It is based on the notion of *Cartesian vectors*, which identifies for a state a sequence of actions that each process can perform without context switches from that state. The intuition behind Cartesian semantics is when Cartesian semantics starts the execution from a state $s$, it selects a sequence of actions $es$ for each process, which are all independent of other process except for the last, and executes them. When the process reaches the target state of the last action in $es$, it starts the procedure again from this state.

**Definition 8 (Cartesian Vector).** *In a concurrent system with $N$ processes $P_1, P_2, \ldots, P_N$, a vector $(p_1, \ldots, p_N) \in Path^N$ is a Cartesian vector from a state $s$ if for each two processes $P_i$, $P_j$ such that $i \neq j$ the following holds:*

- *The first state of $p_i$ is $s$;*
- *$p_i$ is a legal path of process $P_i$;*
- *$\forall t \in p_i, t' \in p_j$: actions $t$ and $t'$ are dependent $\Rightarrow$ $t$ and $t'$ are the last actions of $p_i$ and $p_j$, respectively.*

Since visible actions affect specification states during refinement checking, here we require that partial order reduction is only used for invisible actions in the implementation model. That is, for a Cartesian vector $(p_1, \ldots, p_N)$, if a visible action $\alpha$ exists in any $p_i$ such that $1 \leq i \leq N$, then $\alpha$ must be the last action in $p_i$. Thus, we consider a slightly modified version of the Cartesian function $\phi : S \to Path^N$ to generate a Cartesian vector for each visited state in [27], presented as Algorithm A.1 in Appendix A. The Cartesian semantics generated by $\phi$ is formalized as a binary relation $\to_\phi$, which only relates the last states of Cartesian vectors and is transitively closed.

An important property of Cartesian semantics for linearizability checking is described in the following theorem, which says that if a state that violates linearizability is found with standard semantics, then this state can be found with Cartesian semantics.

**Theorem 6.** *For a Cartesian function $\phi$ and a given state that violates linearizability $(q, \emptyset)$, if $\frac{s*}{\Rightarrow(q,\emptyset)}$, then $s \to_\phi (q, \emptyset)$.*

**Proof.** Given a state that violates linearizability $(q, \emptyset)$, there exists a path $\pi$ of the form $\langle s = s_0, \alpha_1, \alpha_2, \ldots, \alpha_n, s_n = (q, \emptyset)\rangle$. By a simple argument, we know that $\alpha_n$ must be a visible action.

We shall prove that $s \to_\phi q$ by induction on $|\pi|$.

*Basis.* $|\pi| = 0$. The claim holds.

*Induction hypothesis.* Assume that the claim is true for any path $\pi$ such that $|\pi| < n$.

*Induction step:* Consider a path $\pi$ of the form $\langle s = s_0 = (im, sp), \alpha_1, \alpha_2, \ldots, \alpha_n, s_n = (q, \emptyset)\rangle$.

Let $(p_1, p_2, \ldots, p_k)$ be the Cartesian vector that is returned by $\phi(im)$. Suppose that $\alpha_n$ is performed by process $u$ such that $1 \leq u \leq k$. Because $\alpha_n$ is visible, it has to be the last action in $p_u$.

Let $i$ be the first occurrence of an action in $\pi$ which ends a path $p_j$ in $\phi(im)$, i.e., $i \in \{1, \ldots, n\}$ is the smallest number for which there exists $j \in \{1, \ldots, k\}$ such that $\langle s_0, \alpha_1, \ldots, \alpha_i, s_i\rangle$ contains $|p_j|$ transitions of process $P_j$. Thus, each $a_m$ is invisible for $1 \leq m < i$.

We distinguish between the following two cases:

*Case 1.* $\alpha_i = \alpha_n$. In this case, $\alpha_n$ is the only action which ends a path in $\phi(im)$, for any $s \in \pi$. From [27, Lemma 2], we conclude that every two actions of different processes in $\langle s_1, \alpha_1, \ldots, \alpha_i, s_n\rangle$ are independent. Therefore, by successively permuting pairs of adjacent independent transitions we can convert $\pi$ to a new path $s = s_0 \overset{\alpha_1', \alpha_2', \ldots, \alpha_n}{\longrightarrow_\phi} s_n' = (im', sp')$. From [27, Corollary 1], $im' = q$. As all actions except $\alpha_n$ are invisible, $sp' = \emptyset$ and thus $s_n = s_n'$. So $s \to_\phi (q, \emptyset)$.

*Case 2.* $\alpha_i \neq \alpha_n$. From [27, Lemma 2], we conclude that every two actions of different processes in $\langle s_0, \alpha_1, \ldots, \alpha_i, s_i\rangle$ are independent. Therefore, by successively permuting pairs of adjacent independent transitions we can convert $\pi$ to a new path $\langle s = s_0, \alpha_1', \alpha_2', \ldots, \alpha_n', s_n' = (im', sp')\rangle$ that begins with $|p_j|$ actions of $P_j$. From [27, Corollary 1], $im' = q$. Since every permuted action is invisible, $s_n' = s_n$. According to the definition of $\to_\phi$, $s_0 \to_\phi s'_{|p_j|}$. From the induction hypothesis, we get $s'_{|p_j|} \to_\phi s_n$. From pseudotransitivity in [27], $s_0 \to_\phi s_n$, i.e., $s \to_\phi (q, \emptyset)$. □

Furthermore, if a state that violates linearizability is reached by a path in Cartesian semantics, it is straightforward to show this path also exists in standard semantics. Therefore, the following theorem can be established to guarantee the correctness of applying Cartesian partial order reduction to linearizability checking.

**Theorem 7.** *A linearizability violated state is reachable in standard semantics if and only if it is also reachable in Cartesian semantics.*

Algorithm 4.4 is an on-the-fly linearizability checking algorithm with Cartesian partial order reduction. The underlined text shows the differences compared with Algorithm 4.1. Given $X \in S_d$ and an action $\alpha$, the function $exec(X, \alpha)$ returns the successor state of executing $\alpha$ from $X$, i.e., $exec(X, \alpha) = X'$, such that $X \overset{\alpha}{\to}_d X'$. Given a path $\pi = \langle s = s_0, \alpha_1, \alpha_2, \ldots, \alpha_n, s_n\rangle$ of a Cartesian vector $\phi(s)$, we use $lastAction(\pi)$ to denote $\alpha_n$ and $lastState(\pi)$ for $s_n$. To prevent $\phi(s)$ from generating infinite paths, $\phi(s)$ stops extending a path once a loop has been detected and marks such a path as $infinite$. An $infinite$ path from $s$ can only contain invisible actions since any visible action ends the path as required. Therefore, this path is removed from linearizability checking.

**Algorithm 4.4.** Linearizability checking algorithm with partial order reduction

```
Procedure linearizability_por ($L_{im}, L_{sp}$)
1:  $checked := \emptyset$;
2:  $pending.push((init_{im}, \tau^*(init_{sp})))$;
3:  while $pending \neq \emptyset$ do
4:      $(im, sp) := pending.pop()$;
5:      $checked := checked \cup \{(im, sp)\}$;
6:      if $sp = \emptyset$ then
7:          return $false$
8:      end if
9:      for all $\underline{p \in \phi(im)}$ do
10:         if $\underline{p\ is\ not\ marked\ as\ infinite}$ then
11:             $\underline{im' = lastState(p)}$;
12:             $\underline{sp' = exec(sp, lastAction(p))}$;
13:             if $(im', sp') \notin checked$ then
14:             $pending.push((im', sp'))$;
15:         end if
16:     end if
17:   end for
18: end while
19: return $true$
```

As shown above, the key step of this approach is to calculate the Cartesian vector for each visited state. So we describe an execution of Cartesian function $\phi$ from the initial state of the concurrent stack example in the following steps. We refer to the two sequences of transitions found for a state as a Cartesian vector for that state. The transitions enclosed in braces are executed atomically.

At the beginning, because the first actions of both processes are invocation actions, which are visible, each sequence of the Cartesian vector only contains its invocation action.

After finding the two sequences, we nondeterministically pick one of them. For example, suppose we first execute $push\_inv.1$. At the resultant state $(s_3, t_3)$, the Cartesian vector is:

$p_1 : push\_inv.0;$
$p_2 : ss := H; \{H := H + 1; ss := H; \}; push\_res.1.1.$

Again, we nondeterministically pick one of these sequences and execute it entirely without a context switch. Suppose we choose to execute $push\_inv.0$ and get to the state $(s_4, t_4)$. The Cartesian vector is:

$p_1 : ss := H; \{H := H + 1; ss := H; \};$
$p_2 : ss := H$
Notice that the statement in $p_2$ is dependent on the statement $\{H := H + 1; ss := H; \}$ in $p_1$.

If we let process 1 first execute its sequence from $(s_4, t_4)$, then at the resultant state $(s_{29}, t_4)$ the Cartesian vector is:

$p_1 : push\_res.0.1;$
$p_2 : ss := H; \{H := H + 1; ss := H; \}; push\_res.1.2.$

Hence, in this way, the Cartesian semantics saves nine transitions and avoids storing 13 states, indicated by dashed line as illustrated in Fig. 3.

## 4.4 Combining Symmetry Reduction and Partial Order Reduction

The combination of symmetry reduction and partial order reduction was first studied by Emerson et al. [22]. They proposed an abstract framework for combining these two reduction techniques based on both preserving simulation relations, and provided model checking algorithms for LTL-X and CTL*-X formula with simultaneous symmetry and partial order reductions. Iosif [33] later adopted the above algorithms for dynamic programs in which processes and objects are created and destroyed with their on-going executions.

In this section, we present an on-the-fly linearizability checking algorithm (presented in Algorithm 4.5) that combines symmetry reduction and Cartesian partial order reduction simultaneously. There are two main reasons why we design our own approach to combining symmetry reduction and partial order reduction. One is due to the difference between classical temporal logic and refinement model checking. Their checking algorithms are different. Further, the expressive power of LTL and refinement is different. The relationship between refinement checking and LTL model checking has been studied before [36], [39]. On one hand, refinement can specify properties which cannot be specified using LTL, like "an $a$ happens in every other state" [62]. On the other hand, any LTL property can be captured by refinement checking. In [36], Leuschel et al. proposed an translation from LTL to CSP processes via Bücci automata with some special treatment. The downside of this approach is discussed in [44], "this approach is not that useful in practice (because the complexity is on the wrong side of the refinement check for FDR to be efficient, and because it requires several tools to be applied in sequence)." Thus, two methods are not interchangeable; one cannot replace the other. The other reason is that the partial order reduction approach applied in our setting is different from the two related work. We use Cartesian partial order reduction, which [22] and [33] use static approaches based on ample sets. We need to find a Cartesian function that works on the symmetry-reduced LTS instead of an ample function for each state [9]. Due to this diversity, the previous combination approaches do not suit it well for our refinement checking.

**Algorithm 4.5.** Linearizability checking algorithm with symmetry reduction and partial order reduction

```
Procedure linearizability_both $(L_{im}, L_{sp})$
1: $checked := \emptyset$;
2: $pending.push((init_{im}, \tau^*(init_{sp})))$;
3: while $pending \neq \emptyset$ do
4:     $(im, sp) := pending.pop()$;
5:     $checked := checked \cup \{(im, sp)\}$;
6:     if $sp = \emptyset$ then
7:         return $false$
8:     end if
9:     for all$p \in \phi(im)$ do
10:        if $p$ is not marked as infinite
11:            $im' = lastState(p)$;
12:            $(repIm', \langle\gamma_0, \gamma_1, ..., \gamma_n\rangle) = Rep(im')$;
13:            $sp' = exec(sp, lastAction(p))$;
14:            if $\forall 0 \leq i \leq n : (repIm', \gamma_i(sp')) \notin checked$ then
15:                $pending.push((repIm', \gamma_0(sp'))$;
16:            end if
17:        end if
18:    end for
19: end while
20: return $true$
```

The correctness of Algorithm 4.5 is established in the following theorem.

**Theorem 8.** *Algorithm 4.1 finds a linearizability violated state $(q, \emptyset)$ if and only if Algorithm 4.5 finds $(Rep(q), \emptyset)$.*

**Proof (Necessary Condition).** Suppose Algorithm 4.1 finds a path $\pi$ that reaches $s = (q, \emptyset)$. If $|\pi| = 0$, it is trivial that the claim holds. Otherwise, given a path $\pi$ of length $n$ of the form $\langle s_0 = (init_{im}, init_{sp}), \alpha_1, s_1, \alpha_2, \ldots, s_n = (q, \emptyset)\rangle$, we shall prove that there exists a path $\overline{\pi}$ generated by Algorithm 4.5 that reaches $(Rep(q), \emptyset)$.

By Theorem 6, there exists a path $\pi_c$ of the form $\langle s'_0 = (init_{im}, init_{sp}), \alpha'_1, s'_1 = (im_1, sp_1), \alpha'_2, \ldots, s'_n = (q, \emptyset)\rangle$ with Cartesian semantics (but not omitting the intermediate actions and states of legal paths) that reaches $s$. By Theorem 4, there exists a $repPair\text{-}twisted$ path $\pi_r$ of the form $\langle s''_0 = (Rep(init_{im}), Rep(init_{sp})), \alpha'_1, \gamma_1, (Rep(im_1), \gamma_1(sp_1)), \gamma_1(\alpha'_2), \ldots, s''_n = (Rep(q), \emptyset)\rangle$. For any state $s'_k$ of $\pi_c$ that is the first state of some legal path $\langle s'_k, \alpha'_{k+1}, s'_{k+1}, \ldots, \alpha'_{k+t}, s'_{k+t}\rangle$, for state $\gamma(s'_k)$ where $\gamma \in G$, its legal path is $\langle\gamma(s'_k), \gamma(\alpha'_{k+1}), \gamma(s'_{k+1}), \ldots, \gamma(\alpha'_{k+t}), \gamma(s'_{k+t})\rangle$. Then, we can create a path $\overline{\pi}$ of the form $\langle\overline{s_0}, \overline{\alpha_1}, \overline{s_1}, \ldots, \overline{s_n}\rangle$ from $\pi_c$ and $\pi_r$ in the following way: For all $0 \leq i \leq n$, if $s'_i = (im_i, sp_i)$ is the first state of some legal path $\langle s'_i, \alpha'_{i+1}, s'_{i+1}, \ldots, \alpha'_{i+t}, s'_{i+t}\angle$, (in this case $s''_i = (Rep(im_i) = \eta_i(im_i), \eta(sp_i))$ where $\eta = \gamma_i\gamma_{i-1}\ldots\gamma_1$), then for all $0 < m < t$, $\overline{s_{i+m}} = \eta(s'_{i+m})$, $\overline{\alpha_{i+m+1}} = \eta(\alpha'_{i+m+1})$, $\overline{s_i} = s''_i$, and $\overline{s_{i+t}} = s''_{i+t}$ . So $\overline{\pi}$ is a path generated from Algorithm 4.5. Since $s''_n$ must be the last state of some legal path, $\overline{s_n} = s''_n$. Hence, Algorithm 4.5 finds $(Rep(q), \emptyset)$ via $\overline{\pi}$.

TABLE 1
Experiment Results on a Server with Intel Xeon 4-Core CPU*2 and 32 GB Memory

| Algorithm | Process No. | No Reductions | | With SR | | With POR | | With SR & POR | | Gain | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | States | Time | States | Time | States | Time | States | Time | States | Time |
| 4-valued *register* | 3 | 9338 | 0.40 | 4938 | 1.46 | 2443 | 0.38 | 1498 | 0.36 | 84% | 10% |
| 4-valued *register* | 4 | 186316 | 8.96 | 36440 | 26.3 | 32222 | 5.22 | 7845 | 4.00 | 95.9% | 55.4% |
| 4-valued *register* | 5 | 4032362 | 377 | 228560 | 717 | 457101 | 123 | 33944 | 64 | 99.2% | 83% |
| 4-valued *register* | 6 | - | - | 1292755 | 21747 | 7028550 | 3935 | 134461 | 1609 | - | - |
| 4-valued *register* | 7 | - | - | - | - | - | - | 484723 | 53110 | - | - |
| 3-valued *register* | 4 | 52381 | 2.72 | 10764 | 7.42 | 12028 | 1.63 | 3114 | 1.27 | 94.1% | 53.3% |
| 5-valued *register* | 4 | 507082 | 22.7 | 96184 | 73.5 | 71021 | 12.7 | 16377 | 8.66 | 96.8% | 61.9% |
| 6-valued *register* | 4 | 1160625 | 49.2 | 215620 | 195 | 138053 | 34.9 | 30636 | 19.1 | 97.4% | 61.2% |
| 7-valued *register* | 4 | 2352897 | 107 | 430595 | 370 | 245634 | 60.8 | 52908 | 31.4 | 97.8% | 70.7% |
| *counter* of size 4 | 3 | 3674 | 0.31 | 669 | 0.44 | 2910 | 0.36 | 556 | 0.30 | 84.9% | 3.2% |
| *counter* of size 4 | 4 | 124558 | 7.63 | 5935 | 6.93 | 101037 | 11.8 | 4879 | 5.59 | 96.1% | 26.7% |
| *counter* of size 4 | 5 | 5970298 | 673 | 59977 | 409 | 4853123 | 1074 | 53115 | 389 | 99.1% | 42.2% |
| *counter* of size 4 | 6 | - | - | 694620 | 52187 | - | - | 636815 | 53010 | - | - |
| *counter* of size 2 | 5 | 4569968 | 490 | 49947 | 351 | 3766611 | 825 | 42999 | 298 | 99.1% | 39.2% |
| *counter* of size 3 | 5 | 5665598 | 637 | 58083 | 419 | 4645898 | 985 | 51932 | 357 | 99.1% | 44% |
| *counter* of size 4 | 5 | 5970298 | 673 | 59977 | 409 | 4853123 | 1074 | 53115 | 389 | 99.1% | 42.2% |
| *counter* of size 5 | 5 | 6002458 | 659 | 60124 | 398 | 4874975 | 1030 | 53670 | 391 | 99.1% | 40.7% |
| *counter* of size 4 (points) | 3 | 535 | 0.06 | 114 | 0.23 | 198 | 0.11 | 62 | 0.04 | 88.4% | 33.3% |
| *counter* of size 4 (points) | 4 | 9165 | 0.25 | 536 | 0.41 | 2796 | 0.41 | 320 | 0.28 | 96.5% | -12% |
| *counter* of size 4 (points) | 5 | 190367 | 5.40 | 2495 | 9.13 | 52510 | 6.74 | 877 | 4.79 | 99.5% | 11.3% |
| *counter* of size 4 (points) | 6 | 4536325 | 147 | 11221 | 454 | 1165367 | 168 | 4488 | 262 | 99.9% | -78% |
| *counter* of size 4 (points) | 7 | - | - | 48918 | 30184 | - | - | 28652 | 27593 | - | - |
| *counter* of size 2 (points) | 5 | 175747 | 5.08 | 2384 | 8.7 | 48604 | 6.3 | 875 | 4.86 | 99.5% | 4% |
| *counter* of size 3 (points) | 5 | 189527 | 5.72 | 2490 | 8.5 | 52270 | 6.3 | 885 | 4.9 | 99.5% | 14% |
| *counter* of size 5 (points) | 5 | 190367 | 5.35 | 2495 | 8.32 | 52510 | 6.37 | 877 | 4.66 | 99.5% | 13% |
| *queue* of size 3 | 3 | 181591 | 10.2 | 31637 | 14.6 | 76283 | 18.6 | 15267 | 8.45 | 91.6% | 17% |
| *queue* of size 3 | 4 | - | - | 1417457 | 3173 | - | - | 773064 | 2068 | - | - |
| *buggy queue* of size 2 | 3 | 86736 | 4.23 | 74920 | 33 | 74920 | 0.93 | 477 | 0.36 | 99.5% | 91.5% |
| *SNZI* of size 2 | 2 | 17629 | 0.90 | 8824 | 1.6 | 6406 | 1.44 | 3461 | 1.03 | 80.4% | 14.4% |
| *SNZI* of size 3 | 3 | - | - | - | - | - | - | 3524553 | 5224 | - | - |

*Sufficient condition.* Suppose Algorithm 4.5 finds a path $\overline{\pi}$ that reaches $\overline{s} = (Rep(q), \emptyset)$. If $|\overline{\pi}| = 0$, it is trivial that the claim holds. Otherwise, suppose that $\overline{\pi} = \langle \overline{s_0}, \overline{\alpha_1}, \overline{s_1}, \ldots, \overline{\alpha_n}, \overline{s_n} = \overline{s} \rangle$ where $n > 0$. Any state in $\overline{\pi}$ must be either the first state of some legal path, or any state between the first and the last ones exclusively in the path with Cartesian semantics. Pick any state $\overline{s_i}$ that is the first state of some legal path $\pi_i = \langle \overline{s_i}, \overline{\alpha_{i+1}}, \overline{s_{i+1}}, \ldots, \overline{\alpha_{i+t}}, \overline{s_{i+t}} \rangle$; by Lemma 2, there exists a *repPair-twisted* path $\pi_i' = \langle s_0, \alpha_1, \gamma_1, s_1, \ldots, \gamma_i, s_i, \alpha_{i+1}, s_{i+1}, \ldots, \alpha_{i+t}, s_{i+t} \rangle$ where for all $0 \le m \le t$: $s_{i+m} = repPair(\overline{s_{i+m}})$ and for all $0 < k \le t$: $\alpha_{i+k} = repPair(\overline{\alpha_{i+k}})$. Because $\overline{s_i}$ and $\overline{s_{i+t}}$ are the first and last states of $\pi_i$, respectively, $\overline{s_i} = repPair(\overline{s_i}) = s_i$ and $\overline{s_{i+t}} = repPair(\overline{s_{i+t}}) = s_{i+t}$. Then, we replace $\pi_i$ by $\langle s_i, \alpha_{i+1}, s_{i+1}, \ldots, \alpha_{i+t}, s_{i+t} \rangle$. We continue to replace each legal path in $\overline{\pi}$ in this way and get a *repPair-twisted* path that ends at $(Rep(q), \emptyset)$. Therefore, by Theorem 4, $L_{im} \sqsupseteq_T D_{sp}$ has a path that ends at $(q, \emptyset)$. So the claim holds. □

## 5 Experiments

We have implemented our method in the PAT model checker [51] and applied it to a number of concurrent algorithms, including *register*—the K-valued register algorithm[12] in Section 3, *counter*—the concurrent counter algorithm presented in Example 2, *queue*—a concurrent nonblocking queue algorithm in [42, Fig. 3], *buggy queue*—an incorrect queue algorithm [47] and *SNZI*—the first algorithm for scalable nonzero indicators [20]. The processes accessing the concurrent data structures are modeled as calling all possible operations nondeterministically. Table 1 summarizes part of our experiments, where "-" means our implementation ran out of memory, "(points)" means that linearization points are given, and "Gain" means the relative improvement on the number of states and time consumed brought by the combination of symmetry reduction and partial order reduction. All relevant experiment information is available online [1].

From Table 1, we can see that the number of states and running time increase rapidly with data size and the number of processes. This is not surprising because model checking linearizability is in EXPSPACE for both time and space [2]. 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. This is reflected from the stack examples with linearization points in Table 1.

From Table 1, we can see that the number of states and running time increase rapidly with data size and the number of processes. The results conform to theoretical results [2]: 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. This is reflected from the counterexamples with linearization points in Table 1. The

12. We extend this example with multiple $k$ readers and a single writer. The correctness is verified using PAT.

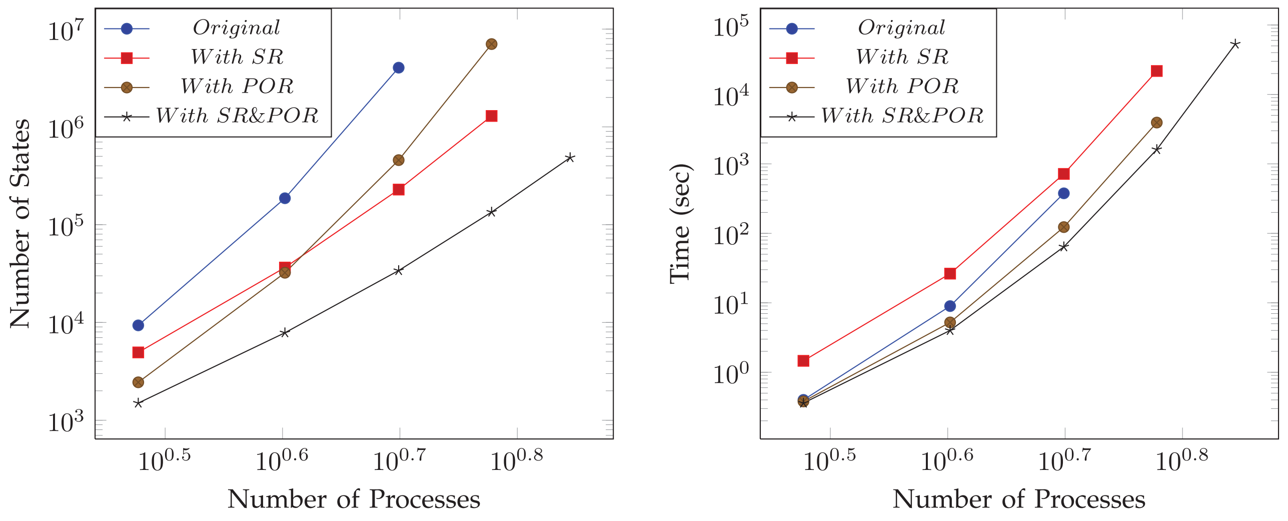

Fig. 4. Performance comparison for the 4-valued *register* algorithm.

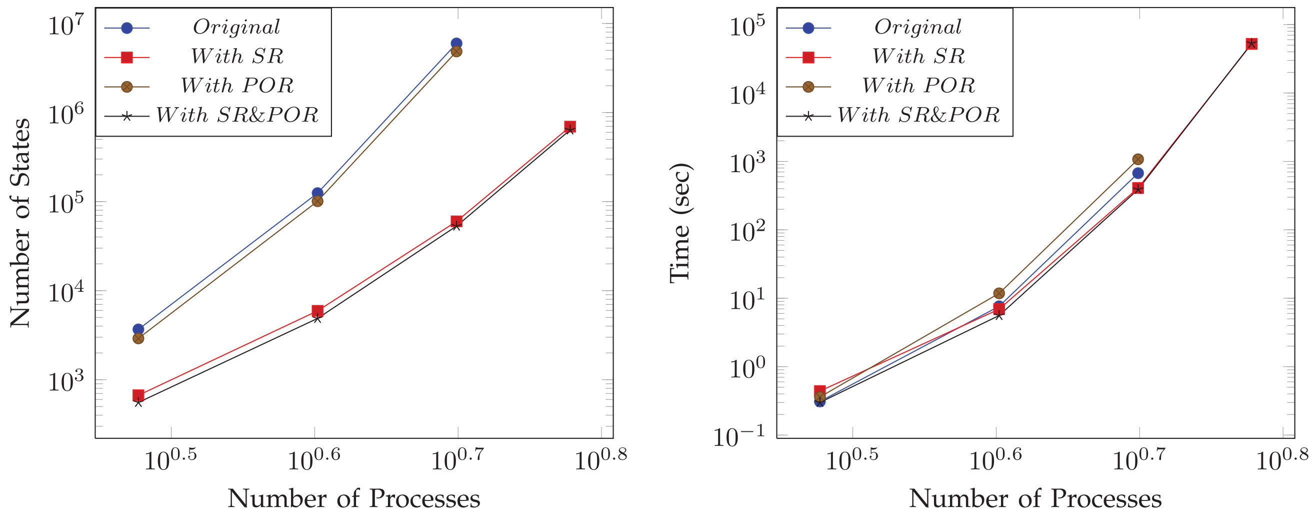

Fig. 5. Performance comparison for the *stack* algorithm of size 4.

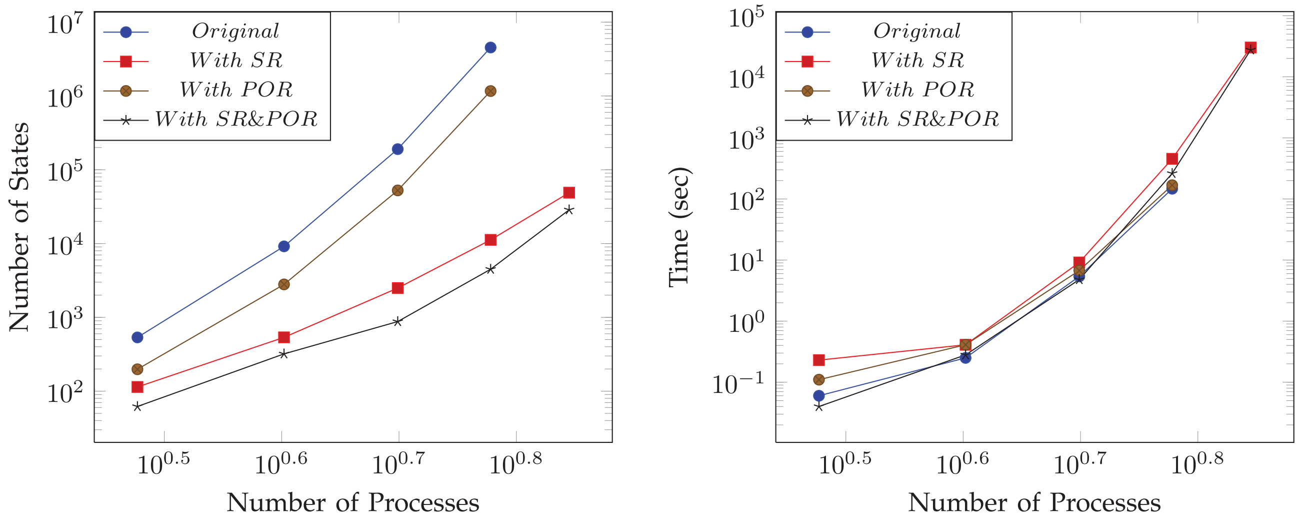

Fig. 6. Performance comparison for the *stack* algorithm of size 4 with given linearization points.

consumed memory and time for the 4-valued *register* algorithm are plotted in Fig. 4, those for the *counter* algorithm of size 4 are in Fig. 5, and those for the *counter* algorithm of size 4 with given linearization points are in Fig. 6. For the case of the same algorithm and the same number of processes, data are not available for some checking algorithms as the memory consumptions for running them were beyond the limit of our server.

As can be seen from the figures, compared with the original algorithm, the rates of increase for used memory by the symmetry reduction, partial order reduction, and their combination algorithms decrease significantly. Symmetry reduction outperforms partial order reduction in line with the increasing number of processes. The combining has the lowest rate of memory increase. Further, it can be seen from the table that the combination reduces the number of states by more than 95 percent on average. Since the bottleneck in this verification task is the memory consumption, the significant improvement brought by the combination postpones the manifestation of the state space explosion problem till deeper levels and therefore allows the verification to complete for some cases that could not complete before, e.g., seven processes for 4-valued *register* and six processes for *counter* of size 7.

On the other hand, the improvement on used time of symmetry reduction, partial order reduction, and their combination is not as desirable as that on used memory. Symmetry reduction sometimes improves the running time and sometimes not. So does partial order reduction. The computational overhead of symmetry reduction stems from checking whether the representative state of the orbit of a visited state has been explored. For each state, we generate all of its automorphisms and pick the lexicographically smallest state as the canonical representative. Thus, calculating canonical representative states is costly if there are a large number of automorphisms. Take the *counter* algorithm with linearization points of size 4 as an example. The 7-process case has seven times the number of automorphisms of the 6-process case, which slows down the linearizability checking a lot. The overhead of partial order reduction is due to dependency analysis between transitions of different processes at each exploration step. Although symmetry reduction (or partial order reduction) by itself does not always improve running time, their combination provides additional improvement, and thus the overall overhead is well compensated by the time we save by combining the two reductions in most cases.

As a result, the combination of both techniques works better than both of them applied in isolation for most cases. This reflects the fact that symmetry and partial order reductions are two orthogonal strategies and can complement each other. The experiments show that our optimization approach can significantly save time and space for demonstrating absence of errors and at the same time it does not sacrifice the capability of detecting bugs.

When the linearizability checking fails, a counterexample trace is returned. In the *buggy queue* example, after analyzing the counterexample trace, it suggests that the dequeued data item is not the first one in the queue, which violates the sequential specification of the queue object.

Experiments suggest that PAT is faster than FDR for systems without variables [49], [61]. Modeling variables using processes and lack of partial order and symmetry reductions will make FDR even slower. Therefore, we skip comparison with FDR on these examples.

## 6 Related Work

Formal verification of linearizability has been studied extensively since linearizability is a central property for the correctness of concurrent algorithms. We discuss and compare with previous approaches in the literature.

*Manual proofs.* Herlihy and Wing [31] coin the notion of linearizability and present a methodology for verifying linearizability by defining a function that maps every state of a concurrent object to the set of all possible abstract values representing it. Vafeiadis et al. [56] show how to apply a rely-guarantee reasoning approach to verifying linearizability for a family of linked list implementations of a set that employ various fine-grained synchronization techniques. Neither approach requires statically determined linearization points, but these manual proofs typically involve a long and repetitive process and require strong expertise on the specific algorithms. Further, there is a great possibility of making subtle mistakes which are difficult to identify.

*Using theorem provers.* Much work has been done on proofs using theorem provers [19], [10], [11], [13], [12]. In these works, Input/Output automata (IOA) are used to model (correct) abstract data structures and concrete implementation algorithms. Linearizability is proven by showing a simulation relation between the abstract automata and implementation automata. The simulation relation is defined in two parts: an abstraction relation relating the abstract and concrete object values, and a step correspondence relating the abstract and concrete program counter values. The proofs have been mechanized using the PVS theorem prover and a number of theories that embody IOA definitions.

Derrick et al. describe a modular approach to establishing linearizability in [15], [14], [16]. Their approach has two parts. First, a generic theory is introduced that encodes linearizability as a special case of data refinement. Local proof obligations for each process are derived based on the theory, and mechanically checked via the KIV theorem prover to make sure that they are sufficient to guarantee linearizability. Second, in practice, a forward simulation relation is built between the concrete and abstract implementations to prove data refinement. How to construct algorithm-specific simulation conditions is demonstrated through the lock-free stack algorithm taken from [10] and the lock-coupling list-based set algorithm taken from [56]. Still, this approach requires that linearization points be statically identified. There are known algorithms which do not satisfy this requirement. This motivates their very recent work [17], which handles the case that a concrete operation mapped to an abstract read-only operation may have linearization points outside the process executing it.

However, theorem prover-based approaches are not fully automatic, e.g., conversion to IOA and use of a theorem prover like PVS require strong expertise. Moveover, they constrain the positions of linearization points and thus cannot be applied to all cases. Therefore, an often cited drawback of theorem provers is that they require a great amount of human effort, which hinders their widespread adoption and usage. On the positive side, such tools can reason about infinite state spaces and complicated data structures in a much more effective way than model checking, such that they can guarantee correctness of an algorithm in all possible scenarios.

*Static analysis.* Amit et al. [3] present a static analysis for verifying linearizability of concurrent unbounded linked data structures using a shape difference abstraction that tracks the difference between two heaps. This approach simultaneously analyzes the concurrent implementation and an executable sequential specification (i.e., a sequential implementation). The two implementations manipulate two disjoint instances of the data structure. The analysis maintains a partial isomorphism between the memory layouts of the two instances. There are several limitations for this approach:

1. every concurrent operation has a (specified) fixed linearization point;
2. the approach verifies linearizability for a fixed but arbitrary number of threads;
3. the approach assumes a garbage collected environment;
4. this approach works well if the concrete heap and the abstract heap have almost identical shapes.

Later, Manevich et al. [41] improved the shape analysis above to handle a larger number of threads. The key idea is to abstract the global heap by decomposing it into (not necessarily disjoint) subheaps, abstracting away some correlations between the subheaps. Decomposition allows reusing subheaps that were decomposed from different heaps, thus representing a set of heaps more compactly (and more abstractly). The resultant algorithm is exponentially faster than the one in [3], being polynomial in the number of threads. Our initial empirical results confirm that our algorithm is able to prove linearizability with 20 threads, 10 times more than in [3]. Recently, Berdine et al. [6] further extend this direction to handle an unbounded number of threads. Their algorithms are based on a new abstract domain whose elements represent thread-quantified invariants, i.e., invariants satisfied by all threads. We exploit existing abstractions to represent the invariants. As a result, our technique lifts existing abstractions by wrapping universal quantification around elements of the base abstract domain.

Vafeiadis [54] further improves this solution to allow linearization points in different threads. Their main limitation, like manual approaches, is that users need to provide linearization points, which are unknown for some algorithms.

This motivates Vafeiadis's work [55] on automating the entire verification process. That work defines the execution of a concrete operation, which maps to the execution of an abstract read-only operation, as a *pure* execution. It assumes that the linearization points with intrinsic conditions or residing within other processes can appear only in a pure execution, similarly to [17]. For each pure execution, a linearizability checker is instrumented into each program point of all processes; for other executions, a checker is instrumented to monitor only the statements of the currently executing process. Then, an abstract interpreter is used to check linearizabilty violations.

*Model checking.* Vechev and Yahav [58] provide two methods for linearizability verification using model checking techniques.

The first method explores, for every history, all possible linearizations, trying to find one that satisfies the sequential specification. Hence, its 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 symmetry reduction, our approach is more scalable than theirs. The second method requires algorithm-specific user annotations for linearization points, which makes it easy to check whether it satisfies the sequential specification. Hence, this method scales better than the first one. However, it is not generic because not all algorithms have explicit linearization points.

Burckhardt et al. [7] present an automatic linearizability checker Line-Up based on the stateless model checker CHESS. Given a deterministic sequential specification, Line-Up is complete but only sound with respect to given inputs. Meanwhile, they generalize the notion of linearizability to handle blocked execution histories.

A new technique designed for concurrent linked-list implementations has been proposed by Cerný et al. [8]. A common pattern in these implementations is that a list entry has a data value from an infinite domain equipped only with the equality and order testing. This pattern makes it possible to represent list content as a data word in automata theory. Cerný et al. reduce the problem of verifying linearizability to the reachability problem of method automata, which simulate how the operations manipulate a concurrent object. They prove that linearizability is decidable for a bounded number of operations. The upside of their approach is that it allows a concurrent object to be stored in a singly linked unbounded heap where the element stored in each location comes from an unbounded data domain; the downside is that it is only capable of checking the executions of two fixed operations due to the severe state explosion problem.

As a coin has two sides, model checking approaches have virtues and limitations. They significantly relax the requirement on user expertise and effort. Most of them do not rely on the knowledge of linearization points nor do they require users to come up with hints to the algorithm in question. The limitation is that the infamous state explosion problem cripples their ability to guarantee the correctness without bounding the data structures and processes in parallel. Our reduction approach clearly does not eliminate the state explosion problem, yet it postpones its manifestation till deeper levels.

In terms of modeling of linearizability, our approach is based on the trace refinement of LTSs, which is similar to [2]. Our refinement checking algorithm is related to existing on-the-fly behavioral equivalence and preorder checking algorithms (e.g., [45], [18]). The nonatomic refinement defined in [14] separates the data explicitly as state-based formalism Object-Z. That modeling requires the knowledge of linearization points, and also prevents automatic verification techniques such as model checking from being used.

## 7 Conclusion

In this work, we studied the formal verification of linearizability using model checking techniques. The key idea of this work is to construct a linearizable specification and express linearizability as the trace refinement relation between the specification and the target concurrent algorithm. Based on whether the linearization points are known or not, we defined two ways of using refinement relations to express linearizability. Based on these definitions, we developed a model checking algorithm for linearizability verification. To further improve the performance, two reduction techniques were proposed, i.e., dynamic partial order reduction and symmetry reduction. Furthermore, we proved that these two reductions are orthogonal to each other and can be applied together. Experimental results showed that our approach is capable of verifying practical algorithms including two recent algorithms: the mailbox problem and scalable nonzero indicators. This work is the first published formal verification for them. The effectiveness of the optimizations is confirmed by the experiments, and combining the two optimizations can achieve better reduction in general. Compared with other model checking linearizability works, our approach requires no knowledge of linearization points, but can significantly speed up the verification if linearization points are given. We have also built the algorithms in a practical model checker, PAT.

The main drawback for our approach is that model checking can only work if the target system has a finite number of states, which limits our approach to working with bounded data structure and finite number of threads. To tackle this problem, several directions are possible. First, one can use abstraction techniques to reduce an infinite number of threads to a small number, e.g., process counter abstraction. The challenge here is how to find the right level of abstraction and detect spurious counterexamples. Second, shape analysis has been proven to be a successful static analysis technique for verifying linearizability. We plan to look at the possibility of incorporating this technique in our model checking to handle unbounded data size. The main idea is to use shape analysis as a kind of data abstraction method to reduce an unbounded data structure to finite shapes. The challenge is to prove that the abstraction is sound.

## Appendix

### Cartesian Function

Let $P_1, P_2, \dots, P_k$ be the processes of the concurrent object algorithms; $\alpha_i$ denotes the action executed by process $P_i$.

**Algorithm A.1.** Algorithm for calculating cartesian vectors on $L_{im}$

```
Procedure $\phi(s)$
1: for all $s \xrightarrow{\alpha_i} s' \in T_{im}$ do
2:     add $\alpha_i$ and $s'$ to $CV[i]$;
3: end for
4: $extendable := \{1, \dots, n\}$;
5: for all $i \in \{1, \dots, n\} : lastAction(CV[i])$ is visible do
6:     $extendable := extendable - \{i\}$;
7: end for
8: for all $i, j \in extendable : i \neq j \wedge lastAction(CV[i])$
   is dependent on last $Action(CV[j])$ do
9:     $extendable := extendable - \{i, j\}$;
10: end for
11: while $extendable \neq \emptyset$ do
12:     pick any $i \in extendable$;
13:     $s := lastState(CV[i])$;
14:     $(\alpha_i, s') := nextTrans(s, i)$;
15:     if $\exists j \neq i : \alpha_i$ is dependent on some action in $CV[j]$
        (other than the last) then
16:         $extendable := extendable - \{i\}$;
17:     else
18:         for all $j \neq i : \alpha'_i$ is dependent on
            last $Action(CV[j])$ do
19:             $extendable := extendable - \{i, j\}$;
20:         end for
21:         if $s' \in CV[i] \wedge i \in extendable$ then
22:             mark $CV[i]$ as $infinite$;
23:             $extendable := extendable - \{i\}$;
24:         end if
25:         if $\alpha_i$ is visible $\wedge\, i \in extendable$ then
26:             $extendable := extendable - \{i\}$;
27:         end if
28:         add $\alpha_i$ and $s'$ to $CV[i]$;
29:     end if
30: end while
31: return $CV$
```

**Helper function:**
$nextTrans(s, i) = (\alpha_i, s') : s \xrightarrow{\alpha_i} s' \in T_{im}$

## Acknowledgments

This work is a substantial extension to [37]. In [37], a proof that linearizability can be verified via refinement checking is established. Further, a model checking algorithm with a classic partial order reduction is proposed and evaluated with concurrent objects. This paper extends [37] in two main aspects: First, our method is generalized to not be limited to any particular kinds of modeling languages; second, this work takes advantage of state-of-the-art optimization techniques to make our method and tool practically useful. In particular, the following optimizations are developed with soundness and completeness proofs: dynamic partial order reduction (which replaces the classic partial order reduction in [37]), symmetry reduction, and their combinations (which have never been explored before). The experimental results (with systems studied in [37] and new ones) show significant improvement with the optimized refinement checking algorithm.

## References

[1] http://www.comp.nus.edu.sg/~pat/linear/, 2013.
[2] R. Alur, K. Mcmillan, and D. Peled, "Model-Checking of Correctness Conditions for Concurrent Objects," *Proc. 12th IEEE Symp. Logic in Computer Science*, pp. 219-228, 1996.
[3] D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav, "Comparison under Abstraction for Verifying Linearizability," *Proc. 19th Int'l Conf. Computer Aided Verification*, pp. 477-490, 2007.
[4] M.K. Arguilera, E. Gafni, and L. Lamport, "The Mailbox Problem," *Proc. 22nd Int'l Symp. Distributed Computing*, pp. 1-15, 2008.

[5] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*, second ed. John Wiley & Sons, Inc., 2004.

[6] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and S. Sagiv, "Thread Quantification for Concurrent Shape Analysis," *Proc. 20th Int'l Conf. Computer Aided Verification,* pp. 399-413, 2008.

[7] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan, "Line-Up: A Complete and Automatic Linearizability Checker," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 330-340, 2010.

[8] P. Cerný, A. Radhakrishna, D. Zufferey, S. Chaudhuri, and R. Alur, "Model Checking of Linearizability of Concurrent List Implementations," *Proc. 22nd Int'l Conf. Computer Aided Verification,* pp. 465-479, 2010.

[9] E.M. Clarke, O. Grumberg, and D.A. Peled, *Model Checking.* MIT Press, 2000.

[10] R. Colvin, S. Doherty, and L. Groves, "Verifying Concurrent Data Structures by Simulation," *Electronic Notes in Theoretical Computer Science,* vol. 137, no. 2, pp. 93-110, 2005.

[11] R. Colvin and L. Groves, "Formal Verification of an Array-Based Nonblocking Queue," *Proc. 10th IEEE Int'l Conf. Eng. of Complex Computer Systems,* pp. 507-516, 2005.

[12] R. Colvin and L. Groves, "A Scalable Lock-Free Stack Algorithm and Its Verification," *Proc. Fifth IEEE Int'l Conf. Software Eng. and Formal Methods,* pp. 339-348, 2007.

[13] R. Colvin, L. Groves, V. Luchangco, and M. Moir, "Formal Verification of a Lazy Concurrent List-Based Set Algorithm," *Proc. 18th Int'l Conf. Computer Aided Verification,* pp. 475-488, 2006.

[14] J. Derrick, G. Schellhorn, and H. Wehrheim, "Proving Linearizability via Non-Atomic Refinement," *Proc. Sixth Int'l Conf. Integrated Formal Methods,* pp. 195-214, 2007.

[15] J. Derrick, G. Schellhorn, and H. Wehrheim, "Mechanizing a Correctness Proof for a Lock-Free Concurrent Stack," *Proc. 10th IFIP WG 6.1 Int'l Conf. Formal Methods for Open Object-Based Distributed Systems,* pp. 78-95, 2008.

[16] J. Derrick, G. Schellhorn, and H. Wehrheim, "Mechanically Verified Proof Obligations for Linearizability," *ACM Trans. Programming Languages and Systems,* vol. 33, no. 1, pp. 4:1-4:43, Jan. 2011.

[17] J. Derrick, G. Schellhorn, and H. Wehrheim, "Verifying Linearisability with Potential Linearisation Points," *Proc. 17th Int'l Conf. Formal Methods,* pp. 323-337, 2011.

[18] D.L. Dill, A.J. Hu, and H. Wong-Toi, "Checking for Language Inclusion Using Simulation Preorders," *Proc. Third Int'l Conf. Computer Aided Verification,* pp. 255-265, 1991.

[19] S. Doherty, L. Groves, V. Luchangco, and M. Moir, "Formal Verification of a Practical Lock-Free Queue Algorithm," *Proc. 24th Int'l Conf. Formal Techniques for Networked and Distributed Systems,* pp. 97-114, 2004.

[20] F. Ellen, Y. Lev, V. Luchangco, and M. Moir, "SNZI: Scalable NonZero Indicators," *Proc. 26th Ann. ACM Symp. Principles of Distributed Computing,* pp. 13-22, 2007.

[21] F. Ellen, Y. Lev, V. Luchangco, and M. Moir, "SNZI: Scalable NonZero Indicators," *Proc. 26th Ann. ACM Symp. Principles of Distributed Computing,* pp. 13-22, 2007.

[22] E. Emerson, S. Jha, and D. Peled, "Combining Partial Order and Symmetry Reductions," *Proc. Third Int'l Workshop Tools and Algorithms for the Construction and Analysis of Systems,* vol. 1217, pp. 19-34, 1997.

[23] E.A. Emerson and A.P. Sistla, "Symmetry and Model Checking," *Formal Methods in System Design,* vol. 9, no. 1/2, pp. 105-131, Aug. 1996.

[24] E.A. Emerson and R.J. Trefler, "From Asymmetry to Full Symmetry: New Techniques for Symmetry Reduction in Model Checking," *Proc. 10th IFIP WG 10.5 Advanced Research Working Conf. Correct Hardware Design and Verification Methods,* pp. 142-156, 1999.

[25] C. Flanagan and P. Godefroid, "Dynamic Partial-Order Reduction for Model Checking Software," *Proc. 32nd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* pp. 110-121, 2005.

[26] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Secaucus, 1996.

[27] G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv, "Cartesian Partial-Order Reduction," *Proc. 14th Int'l SPIN Workshop Model Checking Software,* pp. 95-112, 2007.

[28] T.L. Harris, K. Fraser, and I.A. Pratt, "A Practical Multi-Word Compare-and-Swap Operation," *Proc. 16th Int'l Conf. Distributed Computing,* pp. 265-279, 2002.

[29] D. Hendler, N. Shavit, and L. Yerushalmi, "A Scalable Lock-Free Stack Algorithm," *Proc. 16th Ann. ACM Symp. Parallelism in Algorithms and Architectures,* pp. 206-215, 2004.

[30] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008.

[31] M. Herlihy and J.M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Programming Language and Systems,* vol. 12, no. 3, pp. 463-492, 1990.

[32] J.E. Hopcroft, *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 2001.

[33] R. Iosif, "Symmetry Reduction Criteria for Software Model Checking," *Proc. Ninth Int'l SPIN Workshop Model Checking Software,* vol. 2318, pp. 31-33, 2002.

[34] C.N. Ip and D.L. Dill, "Better Verification through Symmetry," *Formal Methods in System Design,* vol. 9, no. 1/2, pp. 41-75, Aug. 1996.

[35] R.P. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün, "Static Partial Order Reduction," *Proc. Fourth Int'l Conf. Tools and Algorithms for Construction and Analysis of Systems,* pp. 345-357, 1998.

[36] M. Leuschel, T. Massart, and A. Currie, "How to Make FDR Spin LTL Model Checking of CSP by Refinement," *Proc. Int'l Symp. Formal Methods Europe on Formal Methods for Increasing Software Productivity,* pp. 99-118, 2001.

[37] Y. Liu, W. Chen, Y.A. Liu, and J. Sun, "Model Checking Linearizability via Refinement," *Proc. Second World Congress Formal Methods,* pp. 321-337, 2009.

[38] Y. Liu, J. Sun, and J.S. Dong, "PAT 3: An Extensible Architecture for Building Multi-Domain Model Checkers," *Proc. 22nd Ann. Int'l Symp. Software Reliability Eng.,* pp. 190-199, 2011.

[39] G. Lowe, "Specification of Communicating Processes: Temporal Logic versus Refusals-Based Refinement," *Formal Aspects of Computing,* vol. 20, no. 3, pp. 277-294, May 2008.

[40] N. Lynch, *Distributed Algorithms.* Morgan Kaufmann, 1997.

[41] R. Manevich, T. Lev-Ami, M. Sagiv, G. Ramalingam, and J. Berdine, "Heap Decomposition for Concurrent Shape Analysis," *Proc. 15th Int'l Symp. Static Analysis,* pp. 363-377, 2008.

[42] M.M. Michael and M.L. Scott, "Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors," *J. Parallel and Distributed Computing,* vol. 51, pp. 1-26, 1998.

[43] N. Moffat, M. Goldsmith, and B. Roscoe, "A Representative Function Approach to Symmetry Exploitation for CSP Refinement Checking," *Proc. 10th Int'l Conf. Formal Methods and Software Eng.,* pp. 258-277, 2008.

[44] D. Plagge and M. Leuschel, "Seven at One Stroke: LTL Model Checking for High-Level Specifications in b, z, Csp, and More," *Int'l J. Software Tools for Technology Transfer,* vol. 12, no. 1, pp. 9-21, Jan. 2010.

[45] A.W. Roscoe, "Model-Checking CSP," *A Classical Mind: Essays in Honour of C.A.R. Hoare,* pp. 353-378, Prentice Hall,, 1994.

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

[47] C.H. Shann, T.L. Huang, and C. Chen, "A Practical Nonblocking Queue Algorithm Using Compare-and-Swap," *Proc. Seventh Int'l Conf. Parallel and Distributed Systems,* pp. 470-475, 2000.

[48] A.P. Sistla and P. Godefroid, "Symmetry and Reduced Symmetry in Model Checking," *ACM Trans. Programming Languages and Systems,* vol. 26, no. 4, pp. 702-734, July 2004.

[49] J. Sun, Y. Liu, and J.S. Dong, "Model Checking CSP Revisited: Introducing a Process Analysis Toolkit," *Proc. Third Int'l Symp. Leveraging Applications of Formal Methods, Verification and Validation,* vol. 17, pp. 307-322, 2008.

[50] J. Sun, Y. Liu, J.S. Dong, and C.Q. Chen, "Integrating Specification and Programs for System Modeling and Verification," *Proc. Third Int'l Symp. Theoretical Aspects of Software Eng.,* pp. 127-135, 2009.

[51] J. Sun, Y. Liu, J.S. Dong, and J. Pang, "PAT: Towards Flexible Verification under Fairness," *Proc. 21th Int'l Conf. Computer Aided Verification,* pp. 709-714, 2009.

[52] J. Sun, Y. Liu, J.S. Dong, and H.H. Wang, "Specifying and Verifying Event-Based Fairness Enhanced Systems," *Proc. 10th Int'l Conf. Formal Methods and Software Eng.,* pp. 318-337, 2008.

[53] R.K. Treiber, "Systems Programming: Coping with Parallelism," Technical Report RJ 5118, IBM Almaden Research Center, 1986.

[54] V. Vafeiadis, “Shape-Value Abstraction for Verifying Linearizability,” *Proc. 10th Int'l Conf. Verification, Model Checking, and Abstract Interpretation*, pp. 335-348, 2009.

[55] V. Vafeiadis, “Automatically Proving Linearizability,” *Proc. 22nd Int'l Conf. Computer Aided Verification*, pp. 450-464, 2010.

[56] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro, “Proving Correctness of Highly-Concurrent Linearisable Objects,” *Proc. 11th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pp. 129-136, 2006.

[57] A. Valmari, “Stubborn Sets for Reduced State Space Generation,” *Proc. 10th Int'l Conf. Applications and Theory of Petri Nets: Advances in Petri Nets 1990*, pp. 491-515, 1991.

[58] M. Vechev and E. Yahav, “Deriving Linearizable Fine-Grained Concurrent Objects,” *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 125-135, 2008.

[59] M. Vechev, E. Yahav, and G. Yorsh, “Experience with Model Checking Linearizability,” *Proc. 16th Int'l SPIN Workshop Model Checking Software*, pp. 261-278, 2009.

[60] L. Wang and S. Stoller, “Static Analysis of Atomicity for Programs with Non-Blocking Synchronization,” *Proc. 10th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pp. 61-71, 2005.

[61] T. Wang, S. Song, J. Sun, Y. Liu, J.S. Dong, X. Wang, and S. Li, “More Anti-Chain Based Refinement Checking,” *Proc. 14th Int'l Conf. Formal Eng. Methods*, pp. 364-380, 2012.

[62] P. Wolper, “Temporal Logic Can Be More Expressive,” *Proc. 22nd Ann. Symp. Foundations of Computer Science*, pp. 340-348, 1981.

**Yang Liu** received the bachelor's of computing degree in 2005 from the National University of Singapore (NUS), the PhD degree in 2010, and continued with postdoctoral work at NUS. Since 2012, he has been with the Nanyang Technological University as an assistant professor. His research focuses on software engineering, formal methods, and security. Particularly, he specializes in software verification using model checking techniques. This work led to the development of a state-of-the-art model checker, process analysis toolkit.

**Wei Chen** received the bachelor's and master's degrees from the Department of Computer Science and Technology, Tsinghua University, and the PhD degree from the Department of Computer Science, Cornell University, Ithaca, New York. He is a lead researcher at Microsoft Research Asia and an adjunct professor of Tsinghua University. His main research interests include distributed computing, fault tolerance, and social network analysis. He is a member of the IEEE.

**Yanhong A. Liu** received the BS degree from Peking University, the MEng degree from Tsinghua University, and the MS and PhD degrees from Cornell University, Ithaca, New York, all in computer science. She is a professor in the Computer Science Department at Stony Brook University, State University of New York. Her primary research interests are in the areas of programming languages, compilers, and software systems, particularly on general and systematic methods for improving the efficiency of computations.

**Jun Sun** received the bachelor's and PhD degrees in computing science from the National University of Singapore (NUS) in 2002 and 2006, respectively. In 2007, he received the prestigious LEE KUAN YEW postdoctoral fellowship in the School of Computing of NUS. In 2010, he joined the Singapore University of Technology and Design (SUTD) as an assistant professor. He was a visiting scholar at MIT from 2011 to 2012. His research focuses on software engineering and formal methods, in particular, system verification and model checking. He is the cofounder of the PAT model checker.

**Shao Jie Zhang** received the bachelor's degree in software engineering from Northeastern University, China, and is currently working toward the PhD degree in computer science at the National University of Singapore. Her current research interests include software engineering and formal verification, in particular, model checking and state space reduction techniques.

**Jin Song Dong** received the bachelor's and PhD degrees in computing from the University of Queensland, Australia, in 1992 and 1996, respectively. From 1995 to 1998, he was a research scientist at CSIRO in Australia. Since 1998, he has been at the School of Computing of the National University of Singapore (NUS), where he is currently an associate professor and one of the PhD supervisors at the NUS Graduate School. He is on the editorial board of *Formal Aspects of Computing* and *Innovations in Systems and Software Engineering*. His research interests include formal methods, software engineering, pervasive computing, and semantic technologies.