Branching Bisimulation and Concurrent Object Verification

Xiaoxiao Yang*
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
Beijing, China
xxyang@ios.ac.cn

Joost-Pieter Katoen
Software Modeling and Verification Group, RWTH Aachen University
Aachen, Germany
katoen@cs.rwth-aachen.de

Huimin Lin
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
Beijing, China
lhm@ios.ac.cn

Gaoang Liu
State Key Laboratory of Computer Science Institute of Software, Chinese Academy of Sciences
Beijing, China
gaoang@ios.ac.cn

Hao Wu
Software Modeling and Verification Group RWTH Aachen University
Aachen, Germany
hao.wu@informatik.rwth-aachen.de

***Abstract*—Linearizability and progress properties are key correctness notions for concurrent objects. This paper presents novel verification techniques for both property classes. The key of our techniques is based on the branching bisimulation equivalence. We first show that it suffices to check linearizability on the quotient object program under branching bisimulation. This is appealing, as it does not rely on linearization points. Further, by exploiting divergence-sensitive branching bisimilarity, our approach proves progress properties (e.g., lock-, wait-freedom) by comparing the concurrent to-be-verified object program against an abstract program consisting of atomic blocks. Our work thus enables the usage of well-known proof techniques for branching bisimulation to check the correctness of concurrent objects. The potential of our approach is illustrated by verifying linearizability and lock-freedom of 14 benchmark algorithms from the literature. Our experiments confirm one known bug and reveals one new bug.**

***Index Terms*—concurrent data structure, verification, branching bisimulation, lock-free, linearizability**

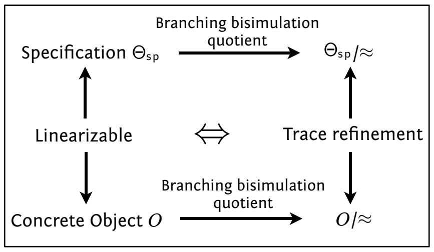
I. Introduction

a) Context: Concurrent data structures such as stacks, queues, hash tables and so forth, are ubiquitous. They are part of packages that come with many programming languages, such as `java.util.concurrent`. Reasoning about scalable concurrent data structures is inherently complex. Threads executing concurrently may interleave yielding different and potentially unexpected outcomes. Advanced synchronization mechanisms such as non-blocking and fine-grained synchronization – pivotal to ensure scalability – further complicate establishing correctness. Commonly accepted aspects of correctness for concurrent data structures include linearizability [18] and progress. In addition to Lamport's notion of sequential consistency [19], linearizability requires that the total ordering which makes it sequentially consistent to respect the "real-time" ordering among the operations in the execution. That is, if an operation e_1 finishes execution before e_2 begins, then e_1 must be ordered before e_2. Linearizability is the key safety property of non-blocking data structures, progress properties such as lock-, wait-freedom [17] address liveness. Lock-free data structures guarantee the progress of at least one thread in each execution. Wait-free data structures ensure that any thread can complete its operation in a finite number of steps.

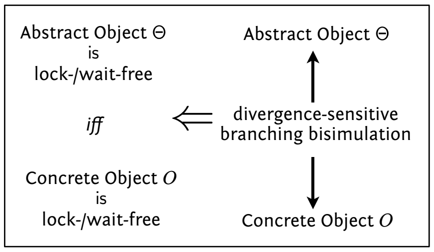
b) This paper: Establishing linearizability and progress properties of scalable concurrent data structures is a highly challenging task. By exploiting the state equivalence relation of object programs, this paper presents a *novel* and *efficient* approach to automatically verify both linearizability and lock-free property. The key to our approach is to exploit the *branching bisimulation* equivalence [32], an elementary equivalence notion in concurrency theory to prove the correctness of an implementation with respect to a (more abstract) specification. We *first* reveal that the linear-time equivalence relation is too coarse to capture the computation effect of a step for non-blocking executions, and branching potentials play a vital role to determine the object state equivalence. This phenomenon is confirmed by the MS lock-free queue [25], a real data structure used in `java.util.concurrent`. We exploit this by (only) viewing method invocations and method returns as visible actions, while considering all other actions to be internal (and invisible). This finding lets us analyze and verify the complex non-blocking algorithms readily, since a lot of inert steps that have no influence on system evolution are abstracted away by branching bisimilarity. The crux of our verification methods is to establish (divergence-sensitive) branching bisimilarity between concurrent data structure implementations and simple abstract data structure descriptions.

c) Checking linearizability.: Before going into more detail about our approach, let us briefly describe the state-of-the-art for checking linearizability. Inspired by the original *linear-time* notion of linearizability by Herlihy and Wing [18],

*Corresponding author.



(a) Proving linearizability without linearization points


Abstract Object Θ is lock-/wait-free
Abstract Object Θ
iff
$\Leftarrow$
divergence-sensitive branching bisimulation
Concrete Object O is lock-/wait-free
Concrete Object O

(b) Proving progress properties

Fig. 1. Verifying linearizability and progress using branching bisimulation.

most existing approaches for verifying linearizability are based on establishing some form of refinement between an abstract specification and a concrete object [3], [9], [20], [23]. These approaches to checking linearizability suffer from two main drawbacks: (1) a complex treatment of linearization points, and (2) the lack of support for proving progress properties. For overlapping method calls, linearizability requires to identify distinct points in their execution intervals – the linearization points (LPs) [17], [20] – such that the happens-before order of concurrent method calls coincides with the order of their LPs. Finding LPs is however complex (the definition is still informal). Verifying non-blocking algorithms with non-fixed LPs such as Heller *et al.*'s lazy set [16] is a hard problem. Techniques exploiting potential LPs require dedicated mechanisms, e.g., [20]. In addition, the refinement approach is not suitable for progress properties; e.g., the refinement technique of [23] of finite-state systems specified as concurrent processes with shared variables, does not preserve progress properties.

d) Verification methods based on branching bisimilarity: Whereas almost all approaches for checking linearizability take a linear-time perspective, we propose to make a paradigm shift, and use a *branching-time* relation instead. We show that branching bisimulation precisely defines the state equivalence for non-blocking object implementation, so it is a natural equivalence relation between a single effective step (e.g., an LP) and a sequence of internal transitions that have the same effect. Based on branching bisimilarity, the paper describes two techniques for verifying correctness properties of concurrent data structures, with several advantages: (1) We can use existing bisimulation checking tools (there are many) to prove linearizability; (2) We can check linearizability on branching bisimulation quotients, resulting in huge state space reductions; (3) Our approach not only does not rely on prior identification of LPs, but can readily analyze the intricate interleavings (based on the internal steps in the quotient system); (4) We can automatically verify lock-freedom in the same framework, using divergence-sensitive branching bisimulation.

Our approaches are summarized in Fig. 1.

To test the efficiency and effectiveness of our approaches, we have conducted a series of experiments on 14 modern highly-optimized concurrent data structures, using the existing proof toolbox CADP [11], originally developed for concurrent systems. A new bug violating lock-freedom was found and a known bug on linearizability was confirmed. Details about verification times, state space sizes are provided in Section 6. To the best of our knowledge, this is the first work which exploits bisimulation techniques to verify linearizability and progress properties on complex concurrent objects and applies state-of-the-art model-checking techniques on such objects.

e) Organization of this paper: Section II briefly reviews abstract and concrete object systems, linearizable specifications, and trace refinements. Section III gives a detailed analysis of the MS lock-free queue and defines state equivalence. Section IV shows the coincidence between max-trace equivalence and branching bisimulation. Section V presents our approaches towards checking linearizability and progress properties. Section VI presents the experiments on the various benchmarks. Section VII analyzes weak bisimulation. Section VIII discusses related work. Section IX concludes.

II. Preliminaries

A. Abstract and concrete objects

A shared object can be a simple variable, or more advanced data structures like stack, linked list etc. There are two kinds of descriptions for concurrent objects: *abstract* and *concrete*. Abstract objects can be regarded as a coarse-grained concurrent implementation, where each method body of object methods is described by one or more atomic blocks. Concrete objects are implemented by using synchronization primitives to refine an atomic operation of abstract objects. Fig. 2 shows the primitive `CAS` (Compare and Swap) that is widely used in non-blocking objects: it compares the value of a memory address `addr` with the expected value `exp` and, if they are the same, updates `addr` with `new` and returns `true`, otherwise returns `false`.

```
Bool CAS(Int& addr, Int exp, Int new) {
  Bool b; Int v;
  atomic{ v:=*addr;
          if (v != exp) {b:=false;}
          else {*addr:=new; b:=true;}
        }
  return b;
}
```

Fig. 2. The primitive Compare and Swap (CAS).

As an example of abstract and concrete objects, let's consider a register with a unique method `NewCompareAndSet` (`NewCAS`) that reads and modifies the register. The abstract implementation of `NewCompareAndSet` is given in Fig. 3.

Instead of a boolean value indicating whether it succeeds, the method returns the register's prior value. A concrete non-blocking implementation based on CAS is shown in Fig. 4, where the method body takes several internal steps to realize a single atomic step of Fig. 3.

```
Int NewCompareAndSet(Int& r, Int exp, Int new){
  Int prior;
  atomic{ prior:=r.get();
          if (prior == exp) *r:=new; }
  return prior;
}
```

Fig. 3. The abstract register method NewCompareAndSet.

```
Int NewCompareAndSet(Int& r, Int exp, Int new){
  Int prior; Bool b:=false;
  while(b == false) {
    prior:=r.get();
    if(prior != exp) return prior;
    else  b:=CAS(r, exp, new);
  }
  return exp;
}
```

Fig. 4. A concrete implementation of NewCompareAndSet.

B. Object systems

The behaviors of a concurrent object can be adequately presented as a labeled transition system. For object methods, we assume there is a programming language equipped with an operational semantics to describe concurrent algorithms and generate the transition system. When analysing and verifying the correctness of a concurrent object, e.g., linearizability [18] and sequential consistency [19], we are only interested in the interactions (i.e., call and return) between the object and its clients, while the internal instructions of the object method are considered invisible, denoted by the silence action τ. Visible actions of an object program have the following two forms:

$$(t, \mathsf{call}, m(n)), \quad (t, \mathsf{ret}(n'), m)$$

where t is a thread identifier and m is a method name, action $(t, \mathsf{call}, m(n))$ is a call action invoking method m by thread t with parameter n, and $(t, \mathsf{ret}(n'), m)$ is a return action of method m by t associated with the return value n'. To generate an object's behaviour, we use *the most general clients* [12], [23], which only repeatedly invoke object's methods in any order and with all possible parameters. We will use the term "object systems" to refer to either the labeled transition systems or the program texts.

Definition 2.1: A labeled transition system (LTS) Δ for a concurrent object is a quadruple $(S, \longrightarrow, \mathcal{A}, s_0)$ where

- S is the set of states,
- $\mathcal{A} = \{(t, \mathsf{call}, m(n)), (t, \mathsf{ret}(n'), m), (t, \tau) \mid t \in \{1 \dots k\}$, where k is the number of threads$\}$ is the set of actions,
- $\longrightarrow \subseteq S \times \mathcal{A} \times S$ is the transition relation,
- $s_0 \in S$ is the initial state. □

We write $s \xrightarrow{a} s'$ to abbreviate $(s, a, s') \in \longrightarrow$, and $s \xrightarrow{\tau} s'$ to mean $s \xrightarrow{(t,\tau)} s'$ for some thread t.

A *path* starting at a state s of an object system is a finite or infinite sequence $s \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} \cdots$. An *execution* is a path starting from the initial state, which represents an entire computation of the object system. A *trace* of state s is a sequence of visible actions obtained from a path of s by omitting states and invisible actions. A *history* is a finite trace starting from the initial state, which models the interactions of a client program with an object.

C. Linearizable specification

Given an object system Δ, we define its corresponding *linearizable specification* [20], [23], denoted by Θ_{sp}, by turning the body of each method in Δ into a single atomic block. A method execution in a linearizable specification Θ_{sp} includes three atomic steps: the call action $(t, \mathsf{call}, m(n))$, the internal action τ (atomic block), and the return action $(t, \mathsf{ret}(n'), m)$. The internal action corresponds to the computation based on the sequential specification of the object. For example, Fig. 3 is a linearizable specification for the implementation of Fig. 4.

D. Trace refinement

Linearizability [18] is a basic safety criterion for concurrent objects, which is defined on histories. Checking linearizability amounts to verifying the trace refinement [9], [20], [23]. Trace refinement is a subset relationship between traces of two object systems, a concrete implementation and the linearizable specification. Let *trace*(Δ) denote the set of all traces in Δ.

Definition 2.2 (Refinement): Let Δ_1 and Δ_2 be two object systems. Δ_1 refines Δ_2, written as $\Delta_1 \sqsubseteq_{tr} \Delta_2$, if and only if *trace*$(\Delta_1) \subseteq$ *trace*(Δ_2). □

The following theorem shows that trace refinement exactly captures linearizability. A proof of the result is given in [23].

Theorem 2.3: Let Δ be an object system and Θ_{sp} the corresponding linearizable specification. All histories of Δ are linearizable if and only if $\Delta \sqsubseteq_{tr} \Theta_{\text{sp}}$. □

III. Characterizing State Equivalence by Max-Trace Equivalence

The key to effective algorithm analysis and verification is to understand when two states are semantically equivalent. Since trace refinement, which is defined using traces, exactly captures linearizability, it is tempting to say that two states of an object system are equivalent if they have the same set of traces. We take Michael-Scott lock-free queue [25] that is used in `java.util.concurrent`, as a real example, to show that an internal τ-step that takes effect for concurrent method calls, cannot be perceived by means of the linear-time equivalence, but by a branching-time equivalence.

A. State equivalence of MS lock-free queue

The concrete algorithm of the queue is shown in Fig. 5. The queue's representation is a linked list, where `Head` points to the first node (a sentinel), and `Tail` points to the last or the penultimate last node. The sentinel node marks a position in the queue, which is never added or removed. The usual way to prove the linearizability of concurrent object is to identify for each method a linearization point (LP) where the method takes effect [17]. For the MS queue, the successful `CAS` at

`Line 8` and `Line 28` are LPs for the successful enqueue and dequeue, and `Line 20` is the LP for the empty queue case, but it is non-fixed, which depends on the future executions (c.f. [20]). The intuition is that, if we read *null* from `h.next` at `Line 20`, but interleavings with other threads before `Line 21` yield a change of `Head` such that the condition on `Head` at `Line 21` fails, then the method has to restart the loop, and the `Line 20` may not be the LP in the new iteration.

```
01enq(v) {
02 local x,t,s,b;
03 x:=new_node(v);
04 while(true) {
05  t:=Tail; s:=t.next;
06  if (t=Tail) {
07   if (s=null) {
08   b:=cas(&(t.next),s,x);
09    if (b) {
10     cas(&Tail,t,x);
11     return true; }
12   }else cas(&Tail,t,s);
13  }
14 }
15}
```

```
16deq() {
17 local h,t,s,v,b;
18 while(true) {
19  h:=Head; t:=Tail;
20  s:=h.next;
21  if (h=Head) {
22   if (h=t) {
23     if (s=null)
24       return EMPTY;
25     cas(&Tail,t,s);
26   }else {
27     v :=s.val;
28     b:=cas(&Head,h,s);
29     if(b) return v;}
30 }
31}
```

Fig. 5. MS lock-free queue: enqueue and dequeue

The intricate interleavings between `Lines 20,21` make trace equivalence no longer adequate to recognize the effect of an essential internal step. This phenomenon is validated by the following instance. Consider an object system involving 2 threads and each invokes the methods enqueue and dequeue for 5 times. A part of the transition graph generated from the system is depicted in Fig. 6, where s_0 is the initial state, and the invocation events of Enq and Deq (i.e., statements at `Lines 1,16`) of a thread t are denoted by (t.call.Enq(v)) and (t.call.Deq), respectively. All internal computation steps of a method are regarded as invisible, that is, the τ-transitions. For the sake of readability, each transition is also marked with the corresponding line number (e.g., L20) in the program text. The states marked with $\circ$ have some additional transitions that are irrelevant to the discussion below and hence omitted.

The interleaved executions of threads t_1 and t_2 from s_0 to s_1 are briefly described as follows: the execution trace is given in the right hand side of Fig. 6, where t_2 has invoked methods Enq and Deq five times and t_1 twice. In the last invocation (t_2.call.Deq) of t_2, the queue state is empty, and t_2 first reads `Tail` and `Head` at `Line 19`, then t_1 starts to invoke Enq(10) and completes it sequentially to reach state s. At state s, t_2 executes `Line 20` and then t_1 starts a new invocation (t_1.call.Deq), but has not taken effect. Thus at state s_1, the queue contains item 10, and τ-transitions of t_1 and t_2 at state s_1 are $s_1 \xrightarrow{\tau} s_3$ labeled with t_1.`L28(true)`, and $s_1 \xrightarrow{\tau} s_2$ labeled with t_2.`L21(true)`. The remaining call actions after s_1 are (t_1.call.Enq(20)), (t_1.call.Enq(30)) and (t_1.call.Deq) by t_1. The subtle interleavings let t_2 response *different return values* in its last invocation. The left hand side of Fig. 6 shows the subsequent executions of s_1 (some intermediate steps are denoted by ellipsis), where

- at s_2, it is easy to see t_2 always returns t_2.RET(EMPTY).
- in $s_1 \Longrightarrow s_5$, since step $s_1 \xrightarrow{t_1.\mathrm{L28}} s_3$ changes the queue state to empty, the later check by t_2 at `Line 21` fails and t_2 restarts a new iteration. So after s_5, t_2 always returns t_2.RET(EMPTY), regardless of the invocations of t_1.
- in $s_3 \Longrightarrow q_1$, t_1 dequeues 10 and enqueues 20 in order. So the later check by t_2 at `Line 21` fails and t_2 restarts. In the new iteration, t_2 returns t_2.RET(20).
- in $s_4 \Longrightarrow r_1$, since before s_4, t_2 has restarted a new iteration and executed `Line 20`, after completing Enq(20) by t_1, t_2 checks `Lines 21,22,23` successfully. Therefore, t_2 returns t_2.RET(EMPTY).
- in $s_4 \Longrightarrow r_2$, t_1 completes Enq(20) and Enq(30) and Deq in order. Since t_1 dequeues 20 successfully, t_2 restart the iteration and return t_2.RET(30).

We use $T^1(s)$ to denote the trace set of s. From the above executions, it is not difficult to see that $T^1(s_1) = T^1(s_3)$. First, all the traces of s_3 are the traces of s_1. Obviously s_2 and s_5 have the same traces, so the trace of s_1 is also the trace of s_3. Thus, s_1 and s_3 are trace equivalent. However, the step $s_1 \xrightarrow{t_1.\mathrm{L28}} s_3$ is an LP that takes effect to change the queue state (i.e., the value of `Head`). Therefore, trace equivalence is not a precise notion to capture the computation effect of the fine-grained executions.

The effect of $s_1 \longrightarrow s_3$ is essentially captured by the branching potentials, i.e., the traces of s_3, s_4 and s_5. For the paths $s_1 \longrightarrow s_2$ and $s_3 \longrightarrow s_4 \longrightarrow s_5$, we have $T^1(s_1) = T^1(s_3)$ and $T^1(s_2) = T^1(s_5)$, but $T^1(s_3) \neq T^1(s_4) \neq T^1(s_5)$. So the traces of intermediate state s_4 on $s_3 \longrightarrow s_4 \longrightarrow s_5$ is overlooked by the path $s_1 \longrightarrow s_2$. Such a branching potential is not taken into account in trace equivalence, but is vital to the computation effect in object systems.

B. *K-trace sets*

From the above discussions, in order to arrive at an adequate notion of state equivalence, we need to consider not only the traces of s_1 and s_3, but also the traces of the intermediate states which lie on their paths. Thus such a notion naturally involves a hierarchy of equivalence relations, each constructed on top of the hierarchies below it. This motivates the following definition that coincides with the k-trace set in [32], where $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$ and $\mathbb{N}$ is the set of natural numbers.

Definition 3.1: For each $k \in \mathbb{N}^\infty$ and each state s, let $T^k(s)$ denote the k-trace set of s. The notions of k-traces and k-trace sets are defined inductively as follows:

1) $T^k(s)$ is the set of all k'-traces of s, for $k' < k$.
2) A k-trace of a state s_0 is obtained from a sequence $(T^k(s_0), a_1,\ T^k(s_1), a_2,\ \cdots,\ a_n, T^k(s_n))$, such that $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_n$ is a path, by replacing all subsequences $(T^k(s_i), a_{i+1}, T^k(s_{i+1}), a_{i+2}, \cdots, a_{i+j+1}, T^k(s_{i+j+1}))$ such that $a_{i+1} = a_{i+2} = \cdots = a_{i+j+1} = \tau$ and $T^k(s_i) = T^k(s_{i+1}) = \cdots = T^k(s_{i+j+1})$ with $T^k(s_i)$.

Two states r and s are *k-trace equivalent*, written $r \equiv_k s$, if $T^k(r) = T^k(s)$; They are *max-trace equivalent*, written $r \equiv s$, if $r \equiv_k s$ for all k. □

It is straightforward to see that $\equiv_k$ and $\equiv$ are equivalence relations. By definition, $T^0(s) = \emptyset$ for every state s, and $T^1(s)$

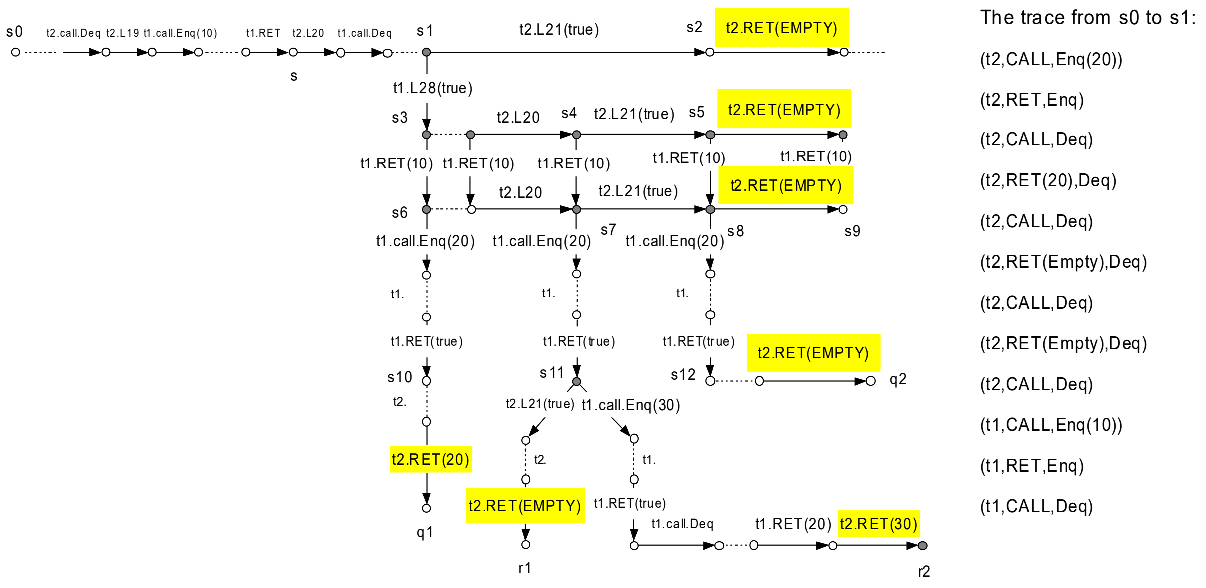


Fig. 6. A part of the transition system of MS lock-free queue: the intricate interleavings in method Deq.

is simply the set of ordinary traces of s. Also if $k' < k$ then $T^{k'}(s) \subseteq T^k(s)$. Note that $r \equiv_k s$ implies $r \equiv_{k'} s$ for any $k' < k$. From this it follows that, for any object system, there exists a k such that $r \equiv_k s$ iff $r \equiv_{k+1} s$. The smallest such a k is called the *cap* of the system.

In an object system, if two states are max-trace equivalent, then they have not only the same ordinary traces (1-traces), but also the same higher traces (k-traces with $k > 1$). Higher traces capture the branching potentials of the states that are passed through in lower traces. This means, as far as visible actions are concerned, there is no way to distinguish the two states in the object system.

Definition 3.2: In an object system, states s and r are equivalent if and only if $s \equiv r$. □

Theorem 3.3: Let Δ_1 and Δ_2 be two object systems such that $\Delta_1 \sqsubseteq_{tr} \Delta_2$. If $s \not\equiv r$ in Δ_1, then $s \not\equiv r$ in Δ_2. □

Example 1: We show that s_1 and s_3 in Fig. 6 are not state equivalent according to Definition 3.2. The 2-trace sets of s_1 and s_3 are as follows:

$$T^2(s_1) = \{(T^1(s_1), \tau, T^1(s_2), \cdots), (T^1(s_1), \tau, T^1(s_3), \cdots), \cdots\}$$
$$T^2(s_3) = \{(T^1(s_3), \tau, T^1(s_4), \tau, T^1(s_5), \cdots), \cdots\}$$

Since $T^1(s_4) \neq T^1(s_3) \neq T^1(s_5)$ and $T^1(s_2) = T^1(s_5)$ and $T^1(s_1) = T^1(s_3)$, it follows $T^2(s_1) \not\subseteq T^2(s_3)$. So $s_1 \not\equiv_2 s_3$.

C. K-trace equivalence of various algorithms

K-trace equivalence not only captures the state equivalence, but also provides a guide to know about the intricacy of non-blocking algorithms. It is normal that in concurrent programs states are not 1-trace equivalent. But systems have states that are 1-trace equivalent but k-trace inequivalent ($k > 1$) imply more intricate executions, like Fig. 6.

We have implemented a tool[1] to check k-trace equivalence between states in a finite system for a given k. The k-trace equivalence of various concurrent algorithms are shown in Table I, where ($\equiv_1$ and $\not\equiv_2$) in the third column means that there exists τ-transition $s \xrightarrow{\tau} r$ in the system such that $s \equiv_1 r$ and $s \not\equiv_2 r$, and $\not\equiv_1$ in the fourth column means that there exists τ-transition $s \xrightarrow{\tau} r$ such that $s \not\equiv_1 r$.

These results indicate that the simple algorithms like Treiber stack and NewCAS, which have fixed LPs, only generate 1-trace inequivalence, but the complicated algorithms (with non-fixed LPs [20]) often involve the "higher" trace inequivalence.

TABLE I
K-TRACE EQUIVALENCE IN VARIOUS CONCURRENT ALGORITHMS.

Object	Non-fixed LPs	$\equiv_1$ and $\not\equiv_2$	$\not\equiv_1$
HW queue [18]	✓	✓	✓
MS queue [25]	✓	✓	✓
DGLM queue [7]	✓	✓	✓
Treiber stack [28]			✓
NewCompareAndSet			✓
CCAS [29]	✓	✓	✓
RDCSS [15]	✓	✓	✓

IV. BRANCHING BISIMULATION FOR CONCURRENT OBJECTS

In concurrency theory, there are various equivalence relations to represent a branching-time equivalence. Branching bisimulation [32] refines Milner's weak bisimulation [34] by requiring two related states should preserve not only their own branching structure but also the branching potentials of all (invisibly reached) intermediate states that are passed through. It turns out that *branching bisimulation exactly captures max-trace equivalence [32]*, in the sense that two state are branching bisimilar if and only if they are max-trace equivalent.

Let $\Longrightarrow$ denote a sequence of zero or more τ-steps. Branching bisimulation for concurrent objects is given as follows.

[1]http://lcs.ios.ac.cn/~xxyang/tools/ktrace-checking.pdf

Definition 4.1: Let $\Delta = (S, \longrightarrow, \mathcal{A}, s_0)$ be an object system. A symmetric relation $\mathcal{R}$ on S is a branching bisimulation if for all $(s_1, s_2) \in \mathcal{R}$, the following holds:

1) if $s_1 \xrightarrow{a} s_1'$ where a is a visible action, then there exists s_2' such that $s_2 \xrightarrow{a} s_2'$ and $(s_1', s_2') \in \mathcal{R}$.
2) if $s_1 \xrightarrow{\tau} s_1'$, then either $(s_1', s_2) \in \mathcal{R}$, or there exist l and s_2' such that $s_2 \Longrightarrow l \xrightarrow{\tau} s_2'$ and $(s_1, l) \in \mathcal{R}$ and $(s_1', s_2') \in \mathcal{R}$. □

Let $\approx \overset{\text{def}}{=} \bigcup\{\mathcal{R} \mid \mathcal{R}$ is a branching bisimulation$\}$. Then $\approx$ is the largest branching bisimulation and an equivalence relation. States s_1 and s_2 are branching bisimilar, if $s_1 \approx s_2$. Two systems are branching bisimilar, if and only if their initial states are branching bisimilar.

In the second clause of the above definition, for $s_2 \Longrightarrow l$ we only require $(s_1, l) \in \mathcal{R}$, without mentioning the states that are visited in $s_2 \Longrightarrow l$. The following Lemma, quoted from [32] guarantees the stuttering property of these intermediate states.

Lemma 4.2: If $r \xrightarrow{\tau} r_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} r_m \xrightarrow{\tau} r'$ is a path such that $r \approx s$ and $r' \approx s$, then $r_i \approx s$ for all i, $1 \leq i \leq m$. □

Thus, the second clause in Definition 4.1 can be alternatively given as:

2. if $s_1 \xrightarrow{\tau} s_1'$, then either $(s_1', s_2) \in \mathcal{R}$, or there exist $l_1, \cdots, l_i$, $i \geq 0$, and s_2' such that $s_2 \xrightarrow{\tau} l_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} l_i \xrightarrow{\tau} s_2'$ and $(s_1, l_1) \in \mathcal{R}, \cdots, (s_1, l_i) \in \mathcal{R}$, $(s_1', s_2') \in \mathcal{R}$.

A discussion on weak bisimulation is given in Section VII.

Theorem 4.3: For any states s and r in an object system, $s \equiv r$ if and only if $s \approx r$. □

It has been shown that max-trace equivalence ($\equiv$) is an adequate notion to capture state equivalence and recognize the essential computation step that takes effect on method calls. However computing max-trace equivalence for a large system is hard. Its equivalent characterization – branching bisimulation ($\approx$) – provides us an efficient way to verify and analyze object systems. For finite-state systems, branching bisimulation equivalence can be checked in polynomial time: the algorithm proposed in [13] has time complexity $O(|\mathcal{A}| + |S| \times |\rightarrow|)$. This result has been recently improved to $O(|{\rightarrow}| \times log|\mathcal{A}| + log|S|)$ in [14].

V. Verifying Linearizability and Lock-Freedom via Branching Bisimulation

In this section, we develop verification methods for proving linearizability and progress properties (e.g., lock-freedom) of concurrent objects based on the (divergence-sensitive) branching bisimulation. In practice, the proposed methods are efficient and convenient for automatically verifying linearizability and lock-free property of finite-state object systems.

A. Verifying linearizability

Given an object system $\Delta = (S, \longrightarrow, \mathcal{A}, s_0)$, for any $s \in S$, let $[s]_\approx$ be the equivalence class of s under $\approx$, and $S/{\approx} = \{[s]_\approx | s \in S\}$ the set of equivalence classes under $\approx$.

Definition 5.1 (Quotient transition system): For an object system $\Delta = (S, \longrightarrow, \mathcal{A}, s_0)$, the quotient transition system $\Delta/{\approx}$ is defined as: $\Delta/{\approx} = (S/{\approx}, \longrightarrow_\approx, \mathcal{A}, [s_0]_\approx)$, where the transition relation $\longrightarrow_\approx$ satisfies the following rules:

$$(1)\frac{s \xrightarrow{\alpha} s'}{[s]_\approx \xrightarrow{\alpha}_\approx [s']_\approx}\ (\alpha \neq \tau) \qquad (2)\frac{s \xrightarrow{\tau} s'}{[s]_\approx \xrightarrow{\tau}_\approx [s']_\approx}\ ((s, s') \not\in \approx)$$

Theorem 5.2: $\Delta/{\approx}$ preserves linearizability. That is, Δ is linearizable if and only if $\Delta/{\approx}$ is linearizable.

Proof: Let Θ_{sp} be the corresponding specification of Δ. Then it is also the corresponding specification of $\Delta/{\approx}$. From Definition 4.1, it is easy to see that $trace(\Delta) = trace(\Delta/{\approx})$. Thus, we have $trace(\Delta) \subseteq trace(\Theta_{\text{sp}})$ iff $trace(\Delta/{\approx}) \subseteq trace(\Theta_{\text{sp}})$. By Definition 2.2, $\Delta \sqsubseteq_{tr} \Theta_{\text{sp}}$ iff $\Delta/{\approx} \sqsubseteq_{tr} \Theta_{\text{sp}}$. Further, by Theorem 2.3, it follows that Δ is linearizable w.r.t. Θ_{sp} iff $\Delta/{\approx}$ is linearizable w.r.t. Θ_{sp}. □

Theorem 5.3: An object system Δ with the corresponding specification Θ_{sp} is linearizable if and only if $\Delta/{\approx} \sqsubseteq_{tr} \Theta_{sp}/{\approx}$. □

It is well-known that deciding trace inclusion is PSPACE-complete, and non-blocking synchronization usually generates a large number of interleavings. Hence verifying linearizability in an automated manner by directly resorting to Theorem 2.3 is infeasible in practice. However, the branching bisimulation quotient is usually much smaller than the object system since it only involves a few steps that are responsible for taking effect for the system. Further, branching bisimulation quotients can be computed efficiently. Thus Theorem 5.3 provides us with a practical solution to the linearizability verification problem:

> Given an object system Δ and a specification Θ_{sp}, first compute their branching bisimulation quotients $\Delta/{\approx}$ and $\Theta_{\text{sp}}/{\approx}$, then check $\Delta/{\approx} \sqsubseteq_{tr} \Theta_{\text{sp}}/{\approx}$.

In practice, this approach results in huge reductions of state spaces. Our experimental results also validate the advantage (c.f. Section VI). Another merit of this method is that verifying linearizability does not rely on prior identifying LPs.

B. Verifying lock-freedom

We exploit *divergence-sensitive* branching bisimulation between a concrete and an abstract object to verify progress properties of concurrent objects. The main result that we will establish is that for divergence-sensitive branching bisimilar abstract and concrete object programs, it suffices to check progress properties on the abstract object program.

Lock-freedom and wait-freedom are the most commonly used progress properties in non-blocking concurrency. Informally, an object implementation is *wait-free* if it guarantees that every thread can finish its started execution in a finite number of steps, while an object implementation is *lock-free* if it guarantees that some thread will complete its execution in a finite number of steps [22]. Their formal definitions specified using next-free LTL are given in [8], [26]. To obtain wait-free object systems, we need to enforce some fairness assumption on object systems to guarantee the fair scheduling of processes. The most common fairness properties (e.g., strong and weak fairness) can all be expressed in next-free LTL. In this section, we focus on the verification method of the lock-free property.

A lock-free object program implies that the entire system always makes progress without infinite τ-paths. An infinite τ-path that does not perform any return action is called *divergent*. To distinguish infinite sequences of internal transitions from finite ones, we treat divergence-sensitive branching bisimulation [1], [35]. Similar definitions are also called branching bisimulation with explicit divergence (e.g., [32]).

Definition 5.4 ([1]): Let $\Delta = (S, \longrightarrow, \mathcal{A}, s_0)$ be an object system and $\mathcal{R}$ an equivalence relation on S.

- A state $s \in S$ is $\mathcal{R}$-divergent if there exists an infinite path $s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots$ such that $(s, s_j) \in \mathcal{R}$ for all $j > 0$.
- $\mathcal{R}$ is divergence-sensitive if for all $(s_1, s_2) \in \mathcal{R}$: s_1 is divergent iff s_2 is divergent. □

Definition 5.5: States s_1 and s_2 in object system Δ are divergent-sensitive branching bisimilar, denoted $s_1 \approx_{div} s_2$, if there exists a divergence-sensitive branching bisimulation $\mathcal{R}$ on Δ such that $(s_1, s_2) \in \mathcal{R}$. □

This notion is lifted to object systems in the standard manner, i.e., object systems Δ_1 and Δ_2 are divergent-sensitive branching bisimilar whenever their initial states are related by $\approx_{div}$ in the disjoint union of Δ_1 and Δ_2.

Lemma 5.6: For an infinite τ-path $\rho = s_1 \xrightarrow{\tau} s_2 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_n \xrightarrow{\tau} s_1$, we have $s_i \approx s_j$ for any states s_i and s_j on ρ.

Proof: We prove $s_i \approx s_{i+1}$ for $(1 \le i < n)$ on ρ by showing $s_i \equiv s_{i+1}$, i.e., for $\forall k$, $s_i \equiv_k s_{i+1}$. The proof proceeds by induction on k. The base case of $k = 1$ is straightforward. Now suppose $s_i \equiv_k s_{i+1}$ for $(1 \le i < n)$. We prove that s_i and s_{i+1} are $(k+1)$-trace equivalent. Since $s_i \xrightarrow{\tau} s_{i+1}$, it follows that any k-trace of s_{i+1} is a k-trace of s_i. Therefore, $T^{k+1}(s_{i+1}) \subseteq T^{k+1}(s_i)$. Since ρ is a τ-loop, where s_{i+1} can reach s_i with only τ-steps and $s_i \equiv_k s_{i+1}$ for $(1 \le i < n)$, we have any k-trace of s_i is also a k-trace of s_{i+1}, that is, $T^{k+1}(s_i) \subseteq T^{k+1}(s_{i+1})$. Hence $s_i \equiv s_{i+1}$. By Theorem 4.3, $s_i \approx s_{i+1}$. □

Lemma 5.7: Let Δ be a finite-state system. There is no infinite τ-path in the quotient $\Delta/\!\approx$. □

Divergence-sensitive branching bisimulation implies (next-free) LTL and CTL*-equivalence [1]. This also holds for countably infinite transition systems that are finitely branching. Thus, $O \approx_{div} \Theta$ implies the preservation of all next-free LTL and CTL*-formulas. Since the lock-freedom (and other progress properties [8]) can be formulated in next-free LTL, for abstract object Θ and concrete object O, it can be preserved by the relation $O \approx_{div} \Theta$.

Theorem 5.8: For the abstract object Θ and concrete object O, if $O \approx_{div} \Theta$, then Θ is lock-free iff O is lock-free. □

For a concrete object, its abstract object is a coarser-grained concurrent implementation. The simplest abstract object program is the concurrent specification specified by one atomic block. For the complicated algorithms such as listed in Table I, the abstract program often needs more than one atomic block. The advantage of this method is that, if an appropriate abstract program for a concrete algorithm can be provided, one can verify the progress on the abstract program, which has a simpler program structure than the concrete algorithm.

However, constructing abstract programs is a manual process. We give another method, which is based on the quotient, to automatically check the lock-freedom of finite systems.

Theorem 5.9: Let Δ be a finite-state object system. If $\Delta \approx_{div} (\Delta/\!\approx)$, then Δ is lock-free iff $(\Delta/\!\approx)$ is lock-free. □

Theorem 5.9 can automatically check divergence. For the finite-state system, if $\Delta \not\approx_{div} (\Delta/\!\approx)$, then by Lemma 5.7, the original system Δ must have a τ-loop, which is a violation of the lock-freedom. For the smaller quotient systems, off-the-shelf model checking tools can be readily applied to check properties such as lock-freedom. Particularly, the single atomic block in the specification Θ_{sp} corresponds to the computation of the sequential specification, which is always assumed to be terminating. It is easy to see that Θ_{sp} is lock-free. Hence, if $(\Delta/\!\approx)$ and Θ_{sp} are trace equivalent, then $(\Delta/\!\approx)$ is lock-free.

VI. Experiments

A. Overview

To illustrate the effectiveness and efficiency of our methods based on branching bisimulation, we have conducted experiments on 14 well-known concurrent data structures using the Construction and Analysis of Distributed Processes (CADP) [11] toolbox [2], including 3 lock-free queues, 4 lists (1 lock-free, 3 lock-based), 4 (lock-free) stacks and 3 extended CAS operations, some of which are used in the `java.util.concurrent` package (e.g., Michael-Scott lock-free queue [25], Harris-Michael lock-free list [17]). The experimental results are summarized in Table II. We found *the new bug* violating lock-free property in the revised version of the Treiber stack in [10] and validated a known bug of HM lock-free list in [17]. To our knowledge, this is the first work which provides a *novel* and *convenient* way to check the linearizability and lock-freedom of concurrent objects and applies branching bisimulation techniques on such objects. One important benefit here is the fully *automated* verification procedure to check the correctness of a concrete object. All experiments run on a server with a 4×12-core AMD CPU @ 2.1 GHz and 192 GB memory under 64-bit Debian 7.6.

TABLE II
VERIFIED ALGORITHMS USING BRANCHING BISIMULATION.

Non-blocking concurrent data structures			
Case study	Linearizability & Lock-freedom	Non-fixed LPs	div.branch.bisim./ trace refine.
1. Treiber stack [28]	✓		✓
2. Treiber stack+ HP [24]	✓		✓
3. Treiber stack+ HP [10]	× Lock-freedom		×
4. MS lock-free queue [25]	✓	✓	✓
5. DGLM queue [7]	✓	✓	✓
6. CCAS [29]	✓	✓	✓
7. RDCSS [15]	✓	✓	✓
8. NewCompareAndSet	✓		✓
9-1. HM lock-free list [17]	× Linearizability	✓	×
9-2. HM lock-free list (revised)	✓	✓	✓
10. HW queue [18]	× Lock-freedom	✓	×
11. HSY stack [37]	✓	✓	✓
Fine-grained concurrent lists			
Case study	Linearizability	Non-fixed LPs	branch.bisim./ trace refine.
12. Heller *et al.* lazy list [16]	✓	✓	✓
13. Optimistic list [17]	✓		✓
14. Fine-grained syn. list [17]	✓		✓

[2]http://cadp.inria.fr/

B. Experimental Setup

To start off, we modeled all concurrent data structures using the LNT modeling language[3], which allows for modeling the object behavior as a set of threads governed by interleaving semantics. CADP provides automated support for generating transition systems of LNT models, obtaining their branching bisimulation quotients, and the verification of temporal logic formulas. When CADP reveals that two transition systems are incomparable (w.r.t. trace refinement or bisimulation), it provides a counterexample. These counterexamples turned out to be very helpful in diagnosing the reason of violation. As the analysis is restricted to *finite* transition systems, we can bound the state space by either: (1) bounding the size of the object (e.g., the size of the stack), or by (2) restricting the number of operations a thread can perform. In the former case, threads can perform operations such as pop and push infinitely many times. We take *the latter approach* since the original algorithms typically do not handle the case of bounded stacks or queues, and imposing such bounds would require amendments of original algorithms for treating overflow situations. Based on these verified models, we conducted the performance analysis of concurrent data structures [40], where a description of the CADP toolsets and the LNT language is given.

C. Verification Results

The experimental results of 14 concurrent objects are summarized in Table II. We discuss the results as follows.

- To verify linearizability, we apply our first approach in Figure 1 (a) (cf. Theorem 5.3) and successfully verify 14 concurrent data structures. In contrast to [3], [20], [36], which rely on linearization points and give different mechanism treating different types of linearization points, our method does not need linearization points, and can support automatic verification.
- To verify progress, our second approach in Figure 1 (b) (cf. Theorems 5.8, 5.9) is applied, and successfully verify 11 algorithms to be lock-free. There are two ways. One way is to manually construct abstract programs and verify their lock-freedom by Theorem 5.8. We construct the abstract programs for MS queue, DGLM queue, CCAS and RDCSS. For static linearization points, the abstract program coincides with the specification. The second way is using the quotient as the abstract system, and automatically verify lock-free property of finite-state systems by Theorem 5.9. Compared to [21], their method proves the lock-free property for 3 non-blocking algorithms, *i.e.,* Treiber stack, MS queue and DGLM queue. They do not discuss more complex algorithms like CCAS, RDCSS and the stacks with Hazard Pointers (HP) [10], [24] as our work.
- Our method is fully-automated for finite systems. Automatic verification allows to find bugs easily if an error diagnostic path reported successfully.

[3]An extension of the ISO standard language LOTOS (ISO:8807:1989)

- Verification based on branching bisimulation equivalence checking for finite state systems (or quotients) is more efficient than trace refinement checking on the original systems, due to the polynomial-time algorithm for checking branching bisimulation. The reduction factor (cf. Figure 10) ranges from at least 5 up to more than 1000 and has an increasing trend if the number of threads and operations per thread is increasing. The verification time for objects can be found in Tables VI, III, IV.

D. A concrete example: the MS lock-free queue

We present the MS lock-free queue as a representative, and show the abstract object and detailed experimental results. The implementation of MS queue can be found in Fig.5.

1) Analyzing algorithms by branching bisimilar: Since the branching bisimulation equivalence captures state equivalence, by computing the quotient, we can easily obtain the essential transitions of concurrent programs. For the MS queue example in Fig. 6, all internal steps in the quotient are labeled with `Lines 8,20,21,28` (Fig. 5). These key statements acquired from the quotient coincide with the manual analysis [20]. Further, we check whether the specification and concrete object are branching bisimilar. If they are not, a path is generated. Fig. 7 is the diagnostic paths involving `Lines 20,28`, which are generated by checking the branching bisimilarity between the quotients of the queue specification and MS queue. This path shows a complicated interleaving of non-fixed LPs, which the specification does not have. From the path, we can refine the specification into finer atomic blocks.

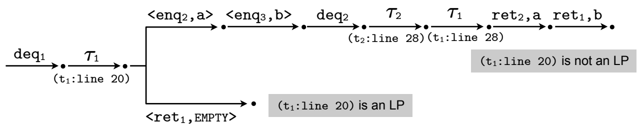


Fig. 7. The non-fixed linearization point in method Deq.

2) The specification and abstract object of MS queue: The specification and abstract queue are shown in Fig. 8. The enqueue method Enq_abs(v) is the same as the specification Enq_spec(v), which has one atomic block. But for dequeue method `Deq_abs`, we need two atomic blocks. The first atomic block `Line 42` matches `Line 20` and the second atomic block `Line 44` matches `Line 28`. In details, atomic block `Line 42` is the linearization point for empty queue such that dequeue returns `EMPTY`, and atomic block `Line 44` is the linearization point for successfully removing the first node from the list. Further, between `Line 42` and `Line 44`, abstract method Deq_abs allows interleavings with other threads that may change the value of `Head`. When atomic block `Line 42` reads `Head` again, of which the value is changed, it will restart the while-loop such that atomic block `Line 42` may not be the linearization point (due to the non-empty queue). Thus, the interleavings of atomic blocks are consistent with the interleavings of `Lines 8,20,21,28` of MS queue.

The DGLM queue [7] that improves the MS queue can be analyzed similarly. By rewriting the concrete program to an equivalent abstract queue, we can verify the much simpler abstract queue instead of the original object program.

```
32 Enq_spec(v){
33  atomic {
      x:=new_node(v);
      t:=Tail;
      t.next:=x;Tail:=x;
34   }
   }


35 Deq_spec() {
36  atomic {
     h:=Head;t:=Tail;
     if(h=t) b:=0
     else
       b:=1;
       s:=h.next;
       Head:=s;
       v:=s.val;
37  }
    if (b==1) return v
    else return EMPTY;
  }
```

```
38 Enq_abs(v){
39  atomic {
      x:=new_node(v);
      t:=Tail;
      t.next:=x;Tail:=x;
40   }
   }

41 Deq_abs() {
    while(true) {
42  atomic {
      h:=Head;t:=Tail;
43  };
44  atomic {
     if(h=Head)
       if(h=t)
         return EMPTY;
       else {
         s:=h.next;
         Head:=s;
         v:=s.val;
         return v;}
     else skip;
45  } } }
```

Fig. 8. Specification `Enq`/`Deq_spec(v)` and abstract queue `Enq`/`Deq_abs(v)`.

3) Experimental results: The verification results for the MS queue as well as the DGLM queue are given in Table VI, where the verification time (in seconds) together with the state spaces of these objects and their quotients are recorded respectively. The MS lock-free and DGLM queues have the same specification (Θ_{SP}) and abstract object (Δ_{Abs}). The DGLM queue has a smaller state-space since it is an optimized version of MS lock-free queue. From Table VI, we can see that the abstract queue and MS/DGLM queues are branching bisimilar, which correspond to the same quotient ($\Delta*/\approx$). These results show that our verification methods are rather efficient to check linearizability and lock-freedom. Although the trace refinement checking in Theorem 5.3 is PSPACE-complete, the quotient system reduces the state space by a factor of 100 times or more, such that verifying linearizablity, which is impossible directly on the original system, becomes possible. For example, for 3 threads with 3 operations, the verification time of about 76 million states takes around only 50 seconds.

E. Automatically verifying lock-freedom

We also provide a full automatic verification method using Theorem 5.9 to check the lock-free property (for finite systems), without constructing abstract objects. The experimental results on checking the lock-free property of the MS queue are presented in Table III, where Δ_{MS} and $\Delta_{\text{MS}}/\approx$ are the state spaces of the original system and the quotient, respectively. From Table III, all the instances of the queue satisfy the lock-free property. Because the quotient is usually much smaller than the abstract object program (c.f. Table VI), the verification method using Theorem 5.9 is more efficient than Theorem 5.8.

Let us see the HM lock-free list that involves the operations add and remove. The experimental results in Table IV indicate that these finite instances of the HM list satisfy the lock-free property. Other algorithms in Table II can be verified similarly.

TABLE III
AUTOMATICALLY CHECKING LOCK-FREEDOM OF THE MS QUEUE [25].

#Th.-#Op.	Δ_{MS}	$\Delta_{\text{MS}}/\approx$	lock-free (Thm.5.9)	time (s)
2-3	49038	863	Yes	0.68
2-4	304049	2648	Yes	1.90
2-5	1554292	6765	Yes	8.44
2-6	7092627	15820	Yes	40.21
3-1	10845	220	Yes	0.52
3-2	1496486	7337	Yes	9.41
3-3	76157266	74551	Yes	516.79

TABLE IV
AUTOMATICALLY CHECKING LOCK-FREEDOM OF THE HM LIST [17].

#Th.-#Op.	Δ_{HM}	$\Delta_{\text{HM}}/\approx$	lock-free (Thm.5.9)	time (s)
2-2	8602	414	Yes	0.44
2-3	55732	1949	Yes	1.01
2-4	227989	5314	Yes	1.96
2-5	670482	10368	Yes	4.29
3-1	16216	445	Yes	0.88

The method dequeue of the HW queue [18] is not lock-free. By comparing the divergence-sensitive branching bisimilarity between the system and its quotient in Table V, we obtain $\Delta_{\text{HW}} \not\approx_{div} (\Delta_{\text{HW}}/\approx)$. The equivalence checking using Theorem 5.9 automatically generates a divergence, shown in Fig. 9. The divergence is found in the `Deq` method.

TABLE V
CHECKING LOCK-FREEDOM OF THE HW QUEUE [18].

#Th.-#Op.	Δ_{HW}	$\Delta_{\text{HW}}/\approx$	lock-free (Thm.5.9)	time (s)
3-1	1324	156	No	0.37

```
*** diagnostic sequence found at depth 2

<initial state>

"CALL !DEQ !1"

"Absent in HW_queue_hide_red .bcg:i - loop (divergence)"
```

Fig. 9. A divergence of the HW queue generated by CADP.

F. Automatic bug hunting

As indicated in Table II, we found two violations of linearizability and lock-freedom. All the found counterexamples are generated in case of just two or three threads. This shows the potential of our approach as bug-hunting technique.

1) We encounter an violation of lock-freedom in the revised Treiber stack of [10]. This revised version prevents the ABA problem from the original Treiber stack but at the expense of violating the wait-freedom of hazard pointers in the original algorithm [24]. We found this bug by an automatically generated counterexample of divergence-sensitive branching bisimulation checking by CADP with just two concurrent threads. The error-path ends in a self-loop in which one thread keeps reading the same hazard pointer value of another thread yielding this thread does not make progress (The detail can be found in [33]).

TABLE VI
VERIFYING LINEARIZABILITY AND LOCK-FREEDOM OF CONCURRENT QUEUES.

#Th.	#Op.	$\Delta_{\mathbf{MS}}$	$\Delta_{\mathbf{DGLM}}$	$\Theta_{\mathbf{SP}}$	$\Delta_{\mathbf{Abs}}$	$\Theta_{\mathbf{SP}}/\approx$	$\Delta*/\approx$	**Theorem 5.8 (lock-free) time (s)**			**Theorem 5.3 (linearizablity) time (s)**		
								MS	DGLM	Result	MS	DGLM	Result
2	1	326	291	72	106	28	28	0.27	0.26	Yes	0.22	0.24	Yes
2	2	5477	4951	855	1325	209	209	0.46	0.43	Yes	0.75	0.70	Yes
2	3	49038	43221	5810	9426	817	863	0.70	0.88	Yes	1.15	1.36	Yes
2	4	304049	261671	30165	50797	2471	2648	2.12	2.17	Yes	2.32	2.31	Yes
2	5	1554292	1310133	136334	237044	6347	6765	9.41	9.05	Yes	4.75	5.02	Yes
2	6	7092627	5881529	571501	1019763	15041	15820	45.23	39.88	Yes	10.79	10.54	Yes
2	7	30265728	24789593	2300270	4187822	33739	35021	210.9	188.27	Yes	27.44	27.16	Yes
3	1	10845	9488	876	1577	220	220	0.49	0.41	Yes	0.76	0.65	Yes
3	2	1496486	1210014	51986	114815	6152	7337	9.35	7.98	Yes	5.20	5.07	Yes
3	3	76157266	55936348	1600195	4110621	62808	74551	613.34	410.91	Yes	54.61	54.79	Yes
4	1	485872	415329	10842	25276	2040	2476	3.32	3.22	Yes	2.18	2.16	Yes

2) Our experiments confirm the (known) bug in the HM lock-free list [17] which was amended in the online errata of [17]. The counterexample is generated by the trace refinement checking of the quotients of concrete object and the specification. It consecutively removes the same item twice, which violates the specification of the list.

G. Efficiency and state-space savings

Fig. 10 shows the overview of the generated state spaces of 11 concurrent data structures and their branching bisimulation quotient systems, where the state-space (x and y-axis) is on logarithmic scale. For the algorithms, we fix the number of threads to 2 and vary the number of operations a thread can perform from 1 to 10. The results show that verifying linearizability based on branching bisimulation quotient is very efficient. For most cases, we obtain a quotient system which is 2 to 3 orders of magnitude (i.e., 100 to 1000 times) smaller than the concrete objects. And in general, for the non-blocking algorithms, the larger the system the higher the state space reduction factor. The largest reductions are obtained for the Treiber stack with hazard pointers (Treiber stack+HP) and the MS lock-free queue yielding a quotient with 0.01% and 0.02% of the size of the concrete objects, respectively.

VII. DISCUSSION ON WEAK BISIMULATION

We give a discussion about weak bisimulation [34]. Weak bisimulation, denoted by $\sim_{\mathsf{w}}$, is obtained by replacing the second clause of Definition 4.1 with:

2. if $s_1 \xrightarrow{\tau} s_1'$, then either $(s_1', s_2) \in \mathcal{R}$, or there exists s_2' such that $s_2 \Longrightarrow\xrightarrow{\tau} s_2'$ and $(s_1', s_2') \in \mathcal{R}$.

Compared with branching bisimulation, weak bisimulation does not require the intermediate states passed through to be matched. Back to the MS queue example in Fig. 6, checking by the CADP tool, it returns $s_1 \sim_{\mathsf{w}} s_3$, along with it $s_2 \not\sim_{\mathsf{w}} s_4$ and $s_2 \sim_{\mathsf{w}} s_5$. For branching bisimulation, the tool reports $s_1 \not\approx s_3$, along with it $s_2 \not\approx s_4$ and $s_2 \approx s_5$.

To explain the difference, consider, for instance, the transition $s_1 \longrightarrow s_2$. In weak bisimulation, it can be matched by $s_3 \longrightarrow s_4 \longrightarrow s_5$, despite that $s_2 \not\sim_{\mathsf{w}} s_4$. However, this is not allowed in branching bisimulation because of $s_2 \not\approx s_4$.

Therefore, branching bisimulation better takes into account the intermediate state that two equivalent states pass through, while weak bisimulation lacks the feature and fails to perceive the effect of linearization point $s_1 \longrightarrow s_3$, which is an essential internal transition for the entire system evolution. So the weak bisimilar is not suitable for analyzing complicated executions.

Another feature of bisimulation is the equivalence checking. For various concurrent data structures, we have checked the equivalence relation $\approx$ and $\sim_{\mathsf{w}}$ between the object system Δ and the corresponding concurrent specification Θ_{sp} respectively. Experimental results in Table VII clearly indicate that, for complex algorithms (e.g., the HSY stack [37]), there does not exist the weak bisimulation equivalence (and also weak bisimulation with explicit divergence) between the system Δ and the specification Θ_{sp} that is specified by one atomic block.

TABLE VII
CHECKING $\Delta \approx \Theta_{\mathrm{sp}}$ AND $\Delta \sim_{\mathsf{w}} \Theta_{\mathrm{sp}}$ FOR VARIOUS ALGORITHMS.

	Object	Δ	$\Delta/\approx$	Θ_{sp}	$\Theta_{\mathrm{sp}}/\approx$	$\sim_{\mathsf{w}}$	$\approx$
2-5	MS	155492	6765	136334	6347	No	No
2-5	DGLM	1310133	6765	136334	6347	No	No
3-2	HW	89483	4018	51986	6152	No	No
3-2	HM	472664	12422	15357	11567	No	No
3-2	Lazy	1311915	15368	15357	11567	No	No
4-1	CCAS	1688	195	822	166	No	No
2-2	Treiber	823	234	487	234	Yes	Yes
3-2	HSY	246761798	11496	12341	5932	No	No

VIII. RELATED WORK

Concurrent object verification including verifying linearizability and lock-freedom has been intensively investigated in the literature. We only discuss the most relevant work.

A plethora of proof-based techniques has been developed for verifying linearizability. Most are based on rely-guarantee reasoning [20], [30], [31], [41], or establishing simulation relations [4], [5], [27]. These techniques often involve identifying linearization points which is a manual non-trivial task. For example, Liang *et al.* [20] propose a program logic tailored to rely-guarantee reasoning to verify complex algorithms. This

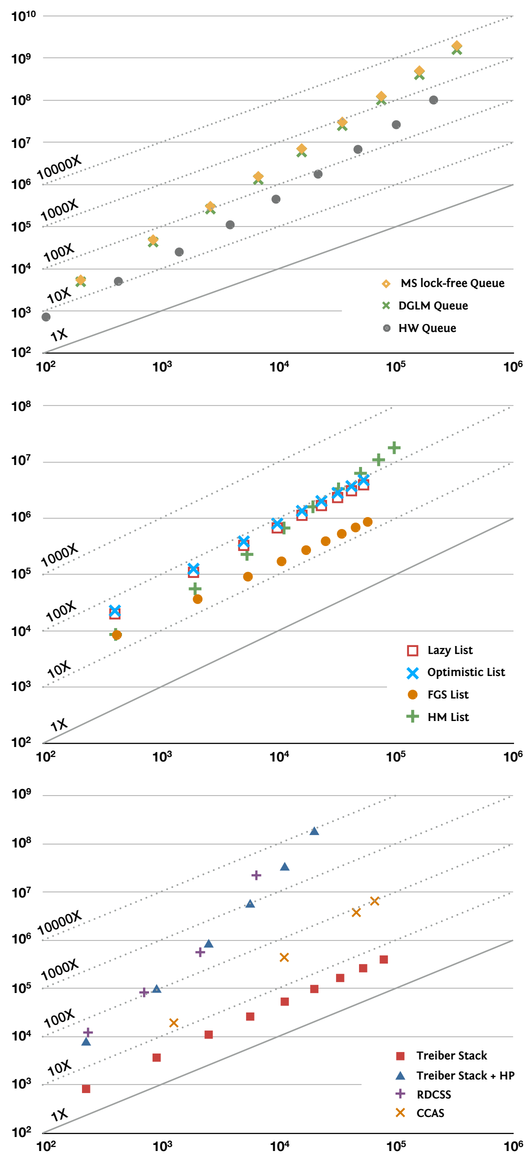


Fig. 10. State-space reduction using ≈-quotienting.

method is applicable to a wide range of popular non-blocking algorithms but is restricted to certain types of linearization points. Challenging algorithms such as [6], [18] fall outside this method. Our techniques do not require identifying linearization points, and take the first step to exploit divergence stuttering/branching bisimulation equivalences [32], [38] from concurrency theory to verify concurrent objects (*The idea was first proposed in 2014* [39]). As we have shown, for finite-state systems, off-the-shelf model checkers can be readily exploited. This also provides a mechanism for finding concurrency bugs, i.e., violations of linearizability and lock-freedom. Although automated verification is not complete, concurrency bugs can be found by treating two to four threads.

Model checking methods to verify linearizability have been proposed in e.g., [2], [3], [23], [36]. Liu et al. [23] formalize linearizability as trace refinement and use partial-order and symmetry reduction techniques to alleviate the state explosion problem. Their experiments are limited to simple concurrent data structures such as counters and registers, and the relation they proposed is not applicable to checking divergence. Our work has modeled and verified various practical algorithms, and can verify both the linearizability and lock-freedom efficiently. Cerny *et al.* [3] propose method automata to verify linearizability of concurrent linked-list implementations, which is restricted to two concurrent threads. An experience report with the model checker SPIN [36] introduces an automated procedure for verifying linearizability, but the method relies on manually annotated linearization points. Some other work (e.g., [2], [42]) employs the stateless model checking method to verify concurrent programs. In particular, [2] presents the automated linearizability checker Line-Up based on the model checker CHESS. Our work uses the branching bisimulation techniques and can capture the effect of certain intricate executions (e.g., LP) of fine-grained concurrent programs (where the linear-time is not sufficient). To the best of our knowledge, all existing algorithms (e.g., [14]) need the entire state space in order to carry out branching bisimulation minimization.

For the verification of progress properties, work [12], [21], [22] recently propose refinement techniques with termination preservation, which can well distinguish divergence from finite traces. However, since refinement notions concern the prefix-closed set of traces, for systems that involve neither divergence nor return actions, refinement-based notions are hard to precisely preserve progress properties. In contrast, the theoretical result of using the notion of divergence-sensitive branching bisimulation can check a large class of progress properties that are expressible in CTL* (containing LTL) without next. In practice, our experiments can treat 11 non-blocking algorithms (finite instances) and found a lock-freedom violation in the revised stack [10]. Some formulations of progress properties using next-free LTL are discussed in [26], [8].

IX. Conclusion

In this paper, we exploit branching bisimulation, denoted by ($\approx$) — a well-established notion in the field of concurrency theory — for proving linearizability and progress properties of concurrent data structures. A concurrent object Δ is linearizable w.r.t. a linearizable specification Θ_{sp} iff their quotients under $\approx$ are in a trace refinement relation. Unlike competitive techniques, this result is independent of the type of linearization points. If the abstract and concrete object are divergence-sensitive branching bisimilar, then progress properties of the — typically much smaller and simpler — abstract object carry over to the concrete object. This entails that lock-freedom (in fact all progress properties that can be expressed in the next-free fragment of CTL*) can be checked on the abstract program. Our approaches can be fully automated for finite-state systems. We have conducted experiments on 14 popular concurrent data structures yielding promising results. In particular, the fact that counterexamples can be obtained in an automated manner is believed to be a useful asset.

Our experiments confirmed a known linearizability bug and revealed a new lock-free property violation.

In addition, due to the precision of branching bisimulation in characterizing state equivalence of object systems, it has the potential to analyze complex non-blocking executions. We have shown that MS lock-free queue can be easily analyzed by this equivalence notion. In the future, we will investigate the further results on analysis of concurrent data structures using branching bisimulation.

Acknowledgment

This research was proposed and done while the first author was at RWTH Aachen University (2013-2015) and supported by the Alexander von Humboldt Foundation. This work was also funded in part by the CDZ project CAP (GZ1023) and NSFC under grant 61100063.

We would like to thank the CONVECS team for the support in CADP, Qiaowen Jia for revising the model of the HSY stack, and anonymous referees for their helpful suggestions on earlier versions of this paper. The first author also thanks Hongfei Fu for his feedback on an earlier version of this work and Wendelin Serwe for his help during the experiments.

References

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[2] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: A Complete and Automatic Linearizability Checker. In *PLDI 2010*, pages 330–340. ACM, 2010.

[3] Pavol Cerný, Arjun Radhakrishna, Damien Zufferey, Swarat Chaudhuri, and Rajeev Alur. Model Checking of Linearizability of Concurrent List Implementations. In *CAV 2010*, LNCS vol.6174, pages 465–479. Springer, 2010.

[4] Robert Colvin, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal Verification of a Lazy Concurrent List-Based Set Algorithm. In *CAV 2006*, LNCS vol.4144, pages 475–488. Springer, 2006.

[5] John Derrick, Gerhard Schellhorn, and Heike Wehrheim. Verifying Linearisability with Potential Linearisation Points. In *FM 2011*, LNCS vol.6664, pages 323–337. Springer, 2011.

[6] Mike Dodds, Andreas Haas, and Christoph M. Kirsch. A scalable, correct time-stamped stack. In *POPL 2015*, pages 233–246, 2015.

[7] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal Verification of a Practical Lock-Free Queue Algorithm. In *FORTE 2004*, LNCS vol.3235, pages 97–114. Springer, 2004.

[8] Brijesh Dongol. Formalising Progress Properties of Non-Blocking Programs. In *ICFEM 2006*, LNCS vol.4260: 284–303. Springer, 2006.

[9] Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for Concurrent Objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010.

[10] Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. Reasoning about Optimistic Concurrency Using a Program Logic for History. In *CONCUR 2010*, LNCS vol.6269, pages 388–402. Springer, 2010.

[11] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2011: a toolbox for the construction and analysis of distributed processes. In *STTT*, 15(2):89–107, 2013.

[12] Alexey Gotsman and Hongseok Yang. Liveness-Preserving Atomicity Abstraction. In *ICALP 2011*, LNCS vol.6756: 453–465. Springer, 2011.

[13] Jan Friso Groote and Frits W. Vaandrager. An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence. In *ICALP 1990*, LNCS vol.443, pages 626–638. Springer, 1990.

[14] Jan Friso Groote, David N. Jansen, Jeroen J. A. Keiren and Anton J.Wijs. An O(m\log n) Algorithm for Computing Stuttering Equivalence and Branching Bisimulation. *ACM Trans. on Computational Logic*. Vol. 18(2), article No. 13. 2017.

[15] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A Practical Multi-Word Compare-and-Swap Operation. In *DISC 2002*, LNCS vol.2508, pages 265–279. Springer, 2002.

[16] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A Lazy Concurrent List-Based Set Algorithm. *Parallel Processing Letters*, 17(4):411–424, 2007.

[17] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[18] Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[19] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. In *IEEE Trans. Comput.*, C-28,9, 690-691. 1979.

[20] Hongjin Liang and Xinyu Feng. Modular Verification of Linearizability with Non-Fixed Linearization points. In *PLDI 2013*, pages 459–470. ACM, 2013.

[21] Hongjin Liang, Xinyu Feng, and Zhong Shao. Compositional Verification of Termination-Preserving Refinement of Concurrent Programs. In *CSL-LICS 2014*, page 65. ACM, 2014.

[22] Hongjin Liang, Jan Hoffmann, Xinyu Feng, and Zhong Shao. Characterizing Progress Properties of Concurrent Objects via Contextual Refinements. In *CONCUR 2013*, LNCS vol.8052, pages 227–241. Springer, 2013.

[23] Yang Liu, Wei Chen, Yanhong A. Liu, Jun Sun, Shao Jie Zhang, and Jin Song Dong. Verifying Linearizability via Optimized Refinement Checking. *IEEE Trans. Software Eng.*, 39(7):1018–1039, 2013.

[24] Maged M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[25] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC 1996*, pages 267–275, 1996.

[26] Erez Petrank, Madanlal Musuvathi, and Bjarne Steensgaard. Progress Guarantee for Parallel Programs via Bounded Lock-Freedom. In *PLDI 2009*, pages 144–154. ACM, 2009.

[27] Gerhard Schellhorn, Heike Wehrheim, and John Derrick. How to Prove Algorithms Linearisable. In *CAV 2012*, pages 243–259, 2012.

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

[29] Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. Logical Relations for Fine-Grained Concurrency. In *POPL 2013*, pages 343–356. ACM, 2013.

[30] Viktor Vafeiadis. Modular Fine-Grained Concurrency Verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory, July 2008.

[31] Viktor Vafeiadis. Automatically Proving Linearizability. In *CAV 2010*, LNCS vol.6174, pages 450–464. Springer, 2010.

[32] Rob J. van Glabbeek and W. P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *J. ACM*, 43(3):555–600, 1996.

[33] Xiaoxiao Yang, Joost-Pieter Katoen, Huimin Lin and Hao Wu. Verifying Concurrent Stacks by Divergence-Sensitive Bisimulation. *Technical Report* at CoRR abs/1701.06104. 2017.

[34] Robin Milner. Communication and Concurrency. Prentice Hall, London, England. 1989.

[35] Rocco De Nicola and Frits W. Vaandrager. Action versus State based Logics for Transition Systems. In *Proceedings of the LITP Spring School on Theoretical Computer Science: Semantics of Systems of Concurrent Processes*, LNCS 469, pages 407-419. Springer, 1990.

[36] Martin T. Vechev, Eran Yahav, and Greta Yorsh. Experience with Model Checking Linearizability. In SPIN 2009, LNCS vol.5578, pages 261Â¨C278. Springer, 2009.

[37] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *SPAA 2004*, pages 206-215, ACM Press. 2004.

[38] K. S. Namjoshi. A Simple Characterization of Stuttering Bisimulation. In *FSTTCS 1997*, LNCS vol.1346, pages 284–296. Springer, 1997.

[39] Xiaoxiao Yang and Joost-Pieter Katoen. Proving Linearizability and Progress of Concurrent Objects by Bisimulation. *Technical Report* ISCAS-SKLCS-14-16. Sep. 2014.

[40] Hao Wu, Xiaoxiao Yang and Joost-Pieter Katoen. Performance Evaluation of Concurrent Data Structures. In *SETTA 2016*, LNCS vol.9984, pages 38-49, Springer, 2016.

[41] Artem Khyzha, Mike Dodds, Alexey Gotsman and Matthew Parkinson. Proving linearizability using partial orders. In *ESOP 2017*, LNCS 10201: 639-667. Springer, 2017.

[42] Jeff Huang. Stateless Model Checking Concurrent Programs with Maximal Causality Reduction. In *PLDI'15*, Vol. 50 (6): 165-174. 2015.