Verifying Concurrent Stacks by Divergence-Sensitive Bisimulation☆

Xiaoxiao Yang[a], Joost-Pieter Katoen[b], Huimin Lin[a], Hao Wu[b]

[a]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
[b]Software Modeling and Verification, RWTH Aachen University, Germany

Abstract

The verification of linearizability – a key correctness criterion for concurrent objects – is based on trace refinement whose checking is PSPACE-complete. This paper suggests to use *branching* bisimulation instead. Our approach is based on comparing an abstract specification in which object methods are executed atomically to a real object program. Exploiting divergence sensitivity, this also applies to progress properties such as lock-freedom. These results enable the use of *polynomial-time* divergence-sensitive branching bisimulation checking techniques for verifying linearizability and progress. We conducted the experiment on the lock-free stacks to validate the efficiency and effectiveness of our methods.

Keywords:
Linearizability, Progress Properties, Bisimulation, Divergence, Concurrency

1. Introduction

Linearizability is a standard correctness condition for the implementation of concurrent objects [1]. Checking linearizability amounts to verify that a concurrent object is implemented correctly by establishing a certain correspondence between the object and its sequential specification. In a multi-threaded environment, however, we also expect that concurrent systems satisfy liveness properties [2], which guarantee that certain "good" events can eventually happen. This includes progress properties [12] such as lock-freedom, wait-freedom and obstruction-freedom, which are introduced to capture the essence of the progress guarantee of non-blocking concurrent algorithms [3]. These algorithms employ fine-grained synchronisations instead of mutex-locks to provide high-performance concurrent implementations of objects, such as Michael-Scott lock-free queue, lock-free stacks [13, 16] and Harris lock-free list (see the `java.util.concurrent` package). Nevertheless, the progress property of an object can affect the progress of the execution of a client program that uses the object. As Filipović *et al.* point out "programmers expect that the behaviour of their program remains the same regardless of whether they use highly efficient data structures or less-optimized but obviously correct data structures.[11]" For optimal use of concurrent objects, it is essential that the basic properties of objects involving linearizability and progress properties are ensured. This emphasizes the importance of developing (automated) techniques for verifying linearizability and progress properties.

The main work on verifying linearizability of concurrent objects is using *trace refinement* checking [5, 8, 10, 11, 21]. This means that 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. Although this approach can be used well to verify linearizability, it is not suitable for verifying progress properties. For example, [8] verifies linearizability based on refinement checking of finite-state systems specified as concurrent processes with shared variables. However, that refinement relation cannot be used to prove progress properties. By the standard definition of linearizability [1], a sequential object specification is employed as the abstract specification, to which any implementation should conform. However, the sequential specification cannot specify liveness. To remedy this, [7] proposes a generalisation of linearizability that can specify liveness while [6] uses atomic blocks to describe the abstract specification where each method body of an object method should be atomic. Although [6, 7] discuss progress properties of objects based on some termination-sensitive contextual refinement, they need to define different refinement relations for progress properties.

Instead of the refinement relation between two objects with different granularities of an atomic operation, an alternative perspective is to capture the relationship by means of appropriate equivalence relation. This is justified by the fact that client programs expect that the observable behaviour of concrete object programs is equivalent to that of abstract ones. The *main* issue is that whether there exists a proper equivalence relation for the object implementation, and if there exists, what kind of equivalence is needed. On the one side, it needs to abstract from internal steps of concrete objects as they typically contain implementation details that are not present in abstract objects. On the other side, we expect that such equivalence preserves some desired properties such that for equivalent abstract and concrete programs, the properties can be checked on the – simpler – abstract program rather than on the concrete program.

We are therefore motivated to explore the equivalence rela-

☆This research was supported by NSFC 61100063 and by a Humboldt Fellowship (X.Y.) from Alexander von Humboldt Foundation.

tion on the externally behavior of objects, and investigate a convenient and efficient method to verify both linearizability *and* progress properties of concurrent objects. Our method employs a tailored version of branching bisimulation [17, 18, 19] to analyze the behaviour of concurrent objects. *Branching bisimulation* is an action-based version of stutter bisimulation [18, 19] that basically allows to abstract from sequences of internal steps. This idea was first proposed in our technical report [9]. In this paper, we present the inspiration and motivation of the idea in [9], and shows the verification method on the non-blocking stacks in detail by directly establishing the divergence-sensitive branching bisimulation between the stack and its specification. Our main contributions are summarized as follows:

- We use the atomic block to specify an abstract object program. We observe that the *effect* of the method call of concrete (fine-grained) objects can be abstracted into the high-level atomic block of the abstract object program such that the fine-grained implementation takes the same effect as the abstract atomic operation. Since the effects of method calls can be equivalently characterized by the observable actions call and return, we show that branching bisimulation is a precise notion to capture the externally observable equivalence between the concrete (fine-grained) object and the abstract (atomic) object. We further show that branching bisimulation preserves linearizability.

- To verify progress properties, we explore the divergence-sensitive equivalence for the abstract and concrete objects. Divergence-sensitive branching bisimulation implies the preservation of $\mathrm{LTL}_{-\bigcirc}$ (the next-free LTL) equivalence, so it preserves various desirable progress properties.

- Instead of trace refinement technique which is computed in PSPACE-complete, our method enables the use of branching bisimulation checking techniques and tools for proving correctness and progress of objects, which can be computed in *polynomial time* for finite systems.

- We successfully apply branching bisimulation with explicit divergence to verify linearizability and lock-free property of non-blocking stacks [16, 27], and reveal a new bug violating the lock-freedom in the revised stack [30]. Thus divergence-sensitive branching bisimulation provides us a unified framework to verify linearizability while preserving progress properties that are specified in $\mathrm{LTL}_{-\bigcirc}$.

Overview. Section 2 defines the object system. Section 3 introduces linearizability and the trace refinement. Section 4 shows branching bisimulation for concurrent objects. Section 5 explores the divergence-sensitive branching bisimulation to verify progress properties. Section 6 conducts the experiment on verifying the non-blocking stacks. Section 7 makes the conclusion.

2. Preliminaries

2.1. Abstract and Concrete Objects

We use two kinds of descriptions for concurrent objects: *abstract* and *concrete*. Abstract objects can be regarded as a concurrent specification, where each method body of every object method is a single atomic operation. Concrete objects involves more intricate interleavings that refines the implementation of abstract objects by replacing the single atomic operation of each method body with fine-grained lock (or lock-free) synchronization technique. For abstract and concrete objects, each method starts with a method call, and ends with a return action.

For instance, an abstract counter with method `inc()` is given in Figure 1 (*i*), where *t* is a local variable and *c* a shared variable, statements embraced by `atomic{ }` is an atomic operation. A concrete counter with the `CAS` instruction is given in Figure 1 (*ii*). The `CAS` instruction allows synchronization between threads at a finer grain.

```
(i) int inc()            (ii) int inc()
    atomic {t:=c;              do t := c;
      c:=t+1;}                 while(CAS(c,t,t+1))
    return;                    return;
```

Figure 1: Abstract counter (*i*) and concrete counter (*ii*).

2.2. Lock-Freedom and Wait-Freedom

Lock-free and wait-free properties are non-blocking progress conditions. Informally, an execution of object methods is *lock-free* if it guarantees that some thread can complete any started operation on the shared data structure in a finite number of steps [12]. An execution is *wait-free* if it guarantees that every thread will complete an operation in a finite number of steps [12].

Suppose that the counter object provides a unique method `inc()` for incrementing a shared variable *c*. Figure 1 (*i*) shows a wait-free implementation in which the increment is done atomically. Figure 1 (*ii*) presents a lock-free implementation by using the `CAS` instruction.

Example 1. Consider the following client program:

```
while (1) do inc(); || inc(); print(z:=0);
```

where ∥ denotes the parallel composition. If the client employs the wait-free counter in Figure 1 (*i*), then the right thread can always terminate and prints $z := 0$ since the wait-freedom of `inc()` guarantees the termination. But if the client uses the lock-free counter in Figure 1 (*ii*), the execution of `inc()` by the right thread may be non-terminating, since the `CAS` instruction in Figure 1 (*ii*) may continuously fail such that the left thread can continuously update shared variable *c*. Thus the right thread never prints $z := 0$. □

In practice, to achieve the desired progress properties of concurrent objects, the garbage collection mechanism with a proper progress condition also needs to be considered.

Example 2. Consider the concurrent stack with hazard pointers [27], where function `Retire_Node` reclaims the released nodes to avoid the ABA-error. To guarantee the lock-free property of the stack, the garbage collector `Retire_Node` needs to be wait-free. However, if `pop` invokes another garbage collector in [30], which is not wait-free and not lock-free, then it will result in the unexpected non-terminating execution of `pop` such that the implementation of the concurrent stack in [30] violates the lock-free property. The detail is shown in Section 6. □

2.3. Object Systems

For analysing concurrent objects, we are interested in the implementation of objects (*i.e.,* object internal actions) and the possible interactions (*i.e.,* call and return) between the client and the object. For clients, object actions are internal and usually regarded as invisible actions, denoted by τ.

Let C be a fixed collection of objects. We have a set of actions Act in the form of:

$$Act ::= \tau \mid (t, \mathsf{call}, o.m(n)) \mid (t, \mathsf{ret}(n'), o.m)$$

where $o \in C$ and t is the thread identifer. Silent action τ is invisible and the other two actions are visible, where $(t, \mathsf{call}, o.m(n))$ means thread t invokes method call method $m(n)$ of object o with parameter n; and $(t, \mathsf{ret}(n), o.m)$ means thread t returns the value n' for method m of object o.

We assume there is a underlying programming language to describe the object program. The language equipped with the operational semantics can generate a state transition system, which is called the *object system.* As in [7, 5], to generate the object behaviour, we assume *the most general client* that repeatedly invokes the object's methods in any order and with all possible parameters.

Definition 2.1 (Object systems). *Let C be a fixed collection of objects. An* object system Δ *is a labelled transition system $(S, \longrightarrow, Act, \varsigma)$, where*

- *S is the set of states;*
- *$Act = \{\tau,\ (t, \mathsf{call},\ o.m(n)),\ (t, \mathsf{ret}(n'),\ o.m) \mid o \in C, t \in \{1 \ldots k\}\}$, where k is the number of threads, Act is the set of actions;*
- *$\longrightarrow \subseteq S \times Act \times S$ is the transition relation;*
- *$\varsigma \in S$ is the initial state.*

Further, we have the following:

- *Abstract object system*, denoted by Θ, is an object system of an abstract object, where the method body of each method is specified by an atomic operation `atomic{}`.
- *Concrete object system*, denoted by O, is an object system of a concrete object, where the method body of each method is specified by the low level synchronization mechanism.
- *Linearizable specification*: for a concrete object system O, we use an abstract object system by turning each method body of O into an atomic block [6, 8], as the corresponding specification, which is called linearizable specification.

In the context, we sometimes use Δ to denote an (abstract or concrete) object system or the corresponding object program.

Each path of the object system is an execution of the object. An *execution fragment* ρ (also called a path fragment) is a finite or infinite alternating sequence of states and actions starting in the initial state s_0 that is $\rho = \{s_0 \alpha_0 s_1 \alpha_1 \cdots \mid (s_i, \alpha_i, s_{i+1}) \in \longrightarrow\}$, where $\alpha_i \in Act$, $s_0 = \varsigma$. A trace is a finite or infinite sequence of observable actions obtained from a path by abstracting away the states and τ-transitions. We shall write $s \xrightarrow{a} s'$ to abbreviate $(s, a, s') \in \longrightarrow$. A state s' is reachable if there exists a finite execution fragment such that $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} s_n = s'$.

3. Linearizability

We briefly introduce the standard linearizability definition [1]. We use a history H, which is a finite sequence of actions call and return, to model an execution of an object system. Given an object system Δ, its set of histories is denoted by $\mathcal{H}(\Delta)$. Subhistory $H \mid t$ of H is the subsequence of all actions in H whose thread name is t. The key idea behind linearizability is to compare concurrent histories to sequential histories.

A history is *sequential* if (1) it starts with a method call, (2) calls and returns alternate in the history, and (3) each return matches immediately the preceding method call. A sequential history is *legal* if it respects the sequential specification of the object. A call is *pending* if it is not followed by a matching return. Let *complete*(H) denote the history obtained from H by deleting all pending calls if the method call has not taken effect yet or adding the corresponding return action if the method call has taken effect.

We use $e.call$ and $e.ret$ to denote, respectively, the invocation and response events of an operation e. An irreflexive partial order $<_H$ on the operations is defined as: $(e, e') \in <_H$ if $e.ret$ precedes $e'.call$ in H. Operations that are not related by $<_H$ are said to be *concurrent* (or overlapping). If H is sequential then $<_H$ is a total order. We first define the linearizability relation between histories.

Definition 3.1 (Linearizability relation between histories). *$H \sqsubseteq_{lin} S$, read "H is linearizable* w.r.t. *S", if (1) S is sequential, (2) $H|t = S|t$ for each thread t, and (3) $<_H \ \subseteq\ <_S$.* □

Thus $H \sqsubseteq_{\mathsf{lin}} S$ if S is a permutation of H preserving (1) the order of actions in each thread, and (2) the non-overlapping method calls in H. Let Γ denote the sequential specification and $\mathcal{H}(\Gamma)$ the set of all histories of Γ. For each $S \in \mathcal{H}(\Gamma)$, S is a legal sequential history. The linearizable object is defined as follows.

Definition 3.2 (Linearizability of object systems). *An object system Δ is* linearizable w.r.t. a sequential specification Γ, *if $\forall H_1 \in \mathcal{H}(\Delta)$. $(\exists S \in \mathcal{H}(\Gamma).\ complete(H_1) \sqsubseteq_{lin} S)$.* □

Linearizability can be casted as the trace refinement [5, 8]. Trace refinement is a subset relationship between traces of two object systems, an implementation and a linearizable specification. Let *trace*(Δ) denote the set of all traces in Δ. Formally, for two object systems Δ_1 and Δ_2, we say Δ_1 refines Δ_2, written as $\Delta_1 \sqsubseteq_{tr} \Delta_2$, if and only if $trace(\Delta_1) \subseteq trace(\Delta_2)$.

Theorem 3.3. *[8]* Let Δ be an object system and Θ the corresponding specification. All histories of Δ are linearizable if and only if $\Delta \sqsubseteq_{tr} \Theta$.

The above theorem shows that trace refinement exactly captures linearizability. A proof of this result can be found in [8].

Remark: Abstract objects with the high-level construct, *i.e.,* atomic blocks, provide us a concurrent specification [6, 7, 8]. Using a sequential specification standard for linearizability, it is necessary to map the concurrent executions to sequential ones limiting our reasoning to these sequential executions [14]. The permutation of concurrent executions to sequential ones is not convenient and even some highly-optimized objects, such as the collision array in the elimination stack [13], do not have the intuitive sequential sequences corresponding to them. Concurrent specification has a more direct and natural relationship with the concurrent implementation. It allows method calls not to terminate (*e.g.,* never return) and the execution of methods to overlap with the executions of others. It adequately models that any method non-termination and overlapping execution interval in the implementation (*i.e.,* concrete object systems) can be reproducible in the specification (*i.e.,* abstract object systems).

4. Branching Bisimulation for Concurrent Objects

By observing the external behavior *call* and *return*, when the abstract object system Θ performs an operation in an atomic step (t, τ) that takes effect for the abstract stack, the concrete one O may take several τ-steps to complete the operation with the same effect as the atomic operation of Θ. This phenomenon results in that the atomic step of Θ can be mimicked by a path with stutter steps of O and a step that takes effect and these stutter steps are exactly right the implementation details of the concrete object, which do not change the state of the object. Vice versa, a path fragment of concrete object system O can also be mimicked by the atomic step of Θ. Such the corresponding relation between Θ and O can be well characterized by a natural equivalence relation, which is an instance of branching bisimulation [17] for concurrent objects. Branching bisimulation is an action-based version of stutter bisimulation [18, 19].

Definition 4.1. Let $\Delta_i = (S_i, \longrightarrow_i, Act_i, \varsigma_i)$ $(i = 1, 2)$ be object systems (abstract or concrete). A symmetric relation $\mathcal{R} \subseteq S_1 \times S_2$ is a *branching bisimulation* for concurrent object, if for any $(s_1, s_2) \in \mathcal{R}$, the following holds:

1. if $s_1 \xrightarrow{a} s_1'$, then there exists s_2' such that $s_2 \xrightarrow{a} s_2'$ and $(s_1', s_2') \in \mathcal{R}$, where a is a visible action $(t, \mathsf{call}, o.m(n))$ or $(t, \mathsf{ret}(n'), o.m)$;
2. if $s_1 \xrightarrow{\tau} s_1'$, then (i) either $(s_1', s_2) \in \mathcal{R}$, (ii) or there exists a finite path fragment $s_2 l_1 \ldots l_k s_2'$ $(k \geq 0)$ such that $s_2 \overset{\tau}{\Longrightarrow} l_k \xrightarrow{\tau} s_2'$ and $(s_1, l_i) \in \mathcal{R}$ $(1 \leq i \leq k)$ and $(s_1', s_2') \in \mathcal{R}$;

where $s_2 \overset{\tau}{\Longrightarrow} l_k$ denotes state l_k is reachable from s_2 by performing zero or more τ-transitions. Δ_1 and Δ_2 are called *branching bisimilar*, denoted $\Delta_1 \sim_{\mathcal{B}} \Delta_2$, if there exists a branching bisimulation $\mathcal{R}$ such that $(\varsigma_1, \varsigma_2) \in \mathcal{R}$. □

Note that the definition of $\sim_{\mathcal{B}}$ can alternatively be given by:

$$\sim_{\mathcal{B}} = \bigcup \{\mathcal{R} \mid \mathcal{R} \text{ is a branching bisimulation with } (\varsigma_1, \varsigma_2) \in \mathcal{R}\}$$

By standard means, we can prove that $\sim_{\mathcal{B}}$ is an equivalence relation, *i.e.,* $\sim_{\mathcal{B}}$ is reflexive, symmetric and transitive.

Relating to the implementation of concurrent objects, the intuitive meaning of Definition 4.1 is explained as follows. Condition (1) states that every outgoing transition of s_1 labelled with a visible action must be matched by an outgoing transition of s_2 with the same visible action. Condition (2) has two cases: (i) for each transition $s_1 \rightarrow s_1'$ labelled with τ, either $(s_1', s_2) \in \mathcal{R}$, which means the step from s_1 to s_1' is a stutter step that does not change the state of the shared object; (ii) or the transition is matched by a path fragment $s_2 l_1 \ldots l_k s_2'$ such that $(s_1', s_2') \in \mathcal{R}$ and $s_2 l_1 l_2 \ldots l_k$ are a series of stutter steps and for each l_i, $(s_1, l_i) \in \mathcal{R}$ (a path is written as $s_0 s_1 \ldots s_n$ for short). Note that stutter steps from s_2 to l_k do not change the object to a new state, but the last step from l_k to s_2' is an atomic step at which point the entire effect of the method call takes place, which corresponds to the execution of the atomic block in the abstract object. After the step that takes effect, the method call may continue to take zero or more stutter steps to complete the remaining method call. Since $\mathcal{R}$ is symmetric, the symmetric counterparts of cases (1)-(2) also apply.

As we see, Definition 4.1 reveals an equivalence relation $\sim_{\mathcal{B}}$ between (concurrent) abstract object Θ and concrete object O, which shifts the *linear-time* paradigm to a *branching-time* paradigm. The *key* point is the condition (2). It is easy to understand the former case that Θ simulates O. For the latter case that O simulates Θ, condition 2-(ii) $(k \geq 0)$ in Definition 4.1 is required, which provides that for each state in Θ, there can be matched by a stutter sequence of τ-steps in O.

Theorem 4.2. Let Θ be an abstract object system. If $O \sim_{\mathcal{B}} \Theta$, then O is linearizable.

Proof: Because $O \sim_{\mathcal{B}} \Theta$, from Definition 4.1, it is easy to see $\sim_{\mathcal{B}}$ preserves trace equivalence, i.e., $trace(O) = trace(\Theta)$. Therefore, $O \sqsubseteq_{tr} \Theta$. By Theorem 3.3, O is linearizable. □

The result shows that linearizability can be proven if we can establish the branching bisimulation equivalence $\sim_{\mathcal{B}}$ between an abstract object and a concrete object.

5. Progress Properties

5.1. Divergence-sensitive branching bisimulation

Divergence means the infinite τ-path of a system. If an object system includes an infinite internal path, *i.e.,* path that only consists of stutter τ-steps, then the path will stay forever in a loop without performing any observable action such as $(t, ret(n), m)$.

We call the stutter path *divergent*. To distinguish infinite series of internal transitions from finite ones, we treat divergence-sensitive branching bisimulation [14, 17].

Definition 5.1. Let $\Delta_i = (S_i, \longrightarrow_i, Act, \varsigma_i)$ be object systems and $\mathcal{R} \subseteq S_1 \times S_2$ an equivalence relation (such as $\sim_{\mathcal{B}}$) on the states of Δ_1 and Δ_2.

- A state u is $\mathcal{R}$-divergent if there exists an infinite path fragment $uu_1u_2\ldots$ such that $(u, u_i) \in \mathcal{R}$ for all $i > 0$.
- $\mathcal{R}$ is divergence-sensitive if for any $(u, v) \in \mathcal{R}$: if u is divergent if and only if v is divergent.

Definition 5.2. If two states s_1 and s_2 in an object system Δ is divergence-sensitive branching bisimilar, denoted by $s_1 \sim_{\mathcal{B}}^{div} s_2$, if there exists a divergence-sensitive branching bisimulation relation $\mathcal{R}$ on Δ such that $(s_1, s_2) \in \mathcal{R}$. □

Two systems Δ_1 and Δ_2 are divergence-sensitive branching bisimilar, if their initial states are related by $\sim_{\mathcal{B}}^{div}$ in the disjoint union of Δ_1 and Δ_2.

5.2. Discussions on divergence of object systems

The notion $\sim_{\mathcal{B}}^{div}$ is a variant of $\sim_{\mathcal{B}}$, only differing in the treatment of divergence. We discuss the ability of these notions $\sim_{\mathcal{B}}$, $\sim_{\mathcal{B}}^{div}$ and $\sqsubseteq_{tr}$ on identifying the divergence of object systems.

Consider the following three counters in Figure 2 associated with methods `inc` and `dec`, where `inc` are the same, whereas `dec` are implemented with distinct progress conditions.

Counter Δ_1:

```
int inc()                int dec()
atomic {c := c+1;}       atomic {c := c-1;}
return;                  return;
```

Counter Δ_2:

```
int inc()                int dec()
atomic {c := c+1;}       while (1) do {skip;}
return;                  return;
```

Counter Δ_3:

```
                         int dec()
int inc()                while (1) do {
atomic {c := c+1;}       if (c>0) then c:=c-1;}
return;                  return;
```

Figure 2: Different progress properties of methods dec().

The `dec` of object Δ_1 is wait-free (and also lock-free) which always guarantees each computation makes progress. The `dec` of Δ_2 does nothing and never makes progress. The `dec` of Δ_3 is a *dependent* progress condition that makes progress only if $c > 0$. This progress condition of Δ_3 is dependent on how the system schedules threads, which is different from the lock-free and wait-free properties of Δ_1.

Assume that thread t_1 invokes `dec` and t_2 invokes `inc` once concurrently. The transition systems for Δ_1, Δ_2 and Δ_3 are partly presented in Figure 3 respectively, where we show the full transition relations from r_3, s_3 and u_3; and other transitions which are not relevant to our discussion are omitted. For convenience, we write "i" to denote τ, and $(t_1, call, dec)$ and (t_1, ret, dec) by dec_1 and ret_1 respectively (it is the same to inc_2). Initially, the counter $c = 0$. Figure 3 (a) shows the executions of Δ_1, where each call by threads t_1 and t_2 can finish the execution in a finite number of steps. In (b), the execution of dec_1 is a self-loop and never returns. In (c), the execution of dec_1 is dependent, where at u_1 and u_3 it is a selp-loop that makes no progress, but after u_4 that the atomic operation τ_2 updates $c = 1$, t_1 evetually returns.

In Figure 3 (b), the self-loop of Δ_2 does not generate the return action ret_1, whereas Δ_1 in (a) can always do the return action ret_1. Therefore, by condition (1) of Definition 4.1, $r_3 \nsim_{\mathcal{B}} s_3$. That is, $\sim_{\mathcal{B}}$ can distinguish the divergent state s_3 in (b) and non-divergent state r_3 in (a).

However, in Figure 3 (c), which is a dependent progress condition, it is easy to see $r_7 \sim_{\mathcal{B}} u_5$, so $r_3 \sim_{\mathcal{B}} u_3$. Thus, for this case, $\sim_{\mathcal{B}}$ is not sufficient to distinguish the self-loop of state u_3 in (c) from the state r_3 in (a). However the notion of divergence-sensitive branching bisimulation $\sim_{\mathcal{B}}^{div}$ can distinguish any divergent state from non-divergent state, that is, $u_3 \nsim_{\mathcal{B}}^{div} r_3$.

For the refinement relation $\sqsubseteq_{tr}$, we found that it cannot distinguish any divergent states, even when the simplest case in Figure 3 (a) and (b), where $trace(s_0) \sqsubseteq_{tr} trace(r_0)$. Therefore relation $\sqsubseteq_{tr}$ is not suitable for checking any progress properties.

5.3. Checking progress properties via bisimulation

There are various progress properties of concurrent objects, such as lock-freedom and wait-freedom for non-blocking concurrency, and starvation-freedom and deadlock-freedom for blocking concurrency. These progress properties can all be specified in $\mathrm{LTL}_{\text{-}\bigcirc}$ (LTL without next) [15, 24].

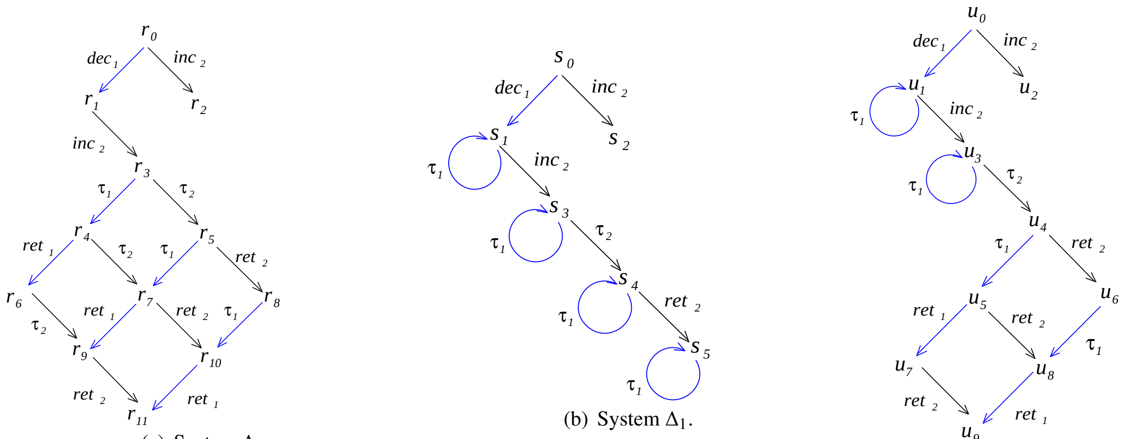
Because the linearizable specification Θ should allow all possible interleavings and be able to terminate the execution of its atomic block at any state, there always exists some thread that can make progress and perform the return action in any execution of Θ. This implies that any execution of abstract object system Θ is lock-free, *i.e.,* any abstract object system Θ is lock-free. Therefore, obtaining a lock-free abstract model is straightforward directly from the concurrent specification.

Lemma 5.3. *The linearizable specification Θ is lock-free.*

Proof: Θ consists of a single atomic block (see Section 3.2), of which the internal execution corresponds to the computation of the sequential specification that by assumption is always terminating. Hence for any run of Θ, there always exists one thread to complete its method call in finite number of steps. □

To obtain wait-free abstract object systems, we need to enforce some fairness constraints to the transition systems to rule out the undesirable paths. The common fair properties (such as strong fairness) can be expressed in next-free LTL [14].

It is known that divergence-sensitive branching bisimulation implies (next-free) $\mathrm{LTL}_{\text{-}\bigcirc}$-equivalence [14]. This also holds for countably infinite transition systems that are finitely branching.



(a) System Δ_1. (b) System Δ_1. (c) System Δ_1.

Figure 3: Divergence-sensitivity of bisimulation for object systems.

Thus, $O \sim_{\mathcal{B}}^{div} \Theta$ implies the preservation of all next-free LTL formulas. Since the lock-freedom (and other progress properties [15, 24]) can be formulated in next-free LTL, for abstract object Θ and concrete object O, the lock-free property can be preserved by the relation $O \sim_{\mathcal{B}}^{div} \Theta$.

Theorem 5.4. Let O be a concrete object system, and Θ the linearizable specification of O.

1. If $O \sim_{\mathcal{B}}^{div} \Theta$, then O is linearizable and lock-free.
2. If $O \sim_{\mathcal{B}}^{div} \Theta$ and Θ is wait-free, then O is linearizable and wait-free.

Proof: Because $O \sim_{\mathcal{B}}^{div} \Theta$, from Definitions 5.1 and 5.2, it is easy to see $\sim_{\mathcal{B}}^{div}$ preserves trace equivalence, i.e., $trace(O) = trace(\Theta)$. Therefore, $O \sqsubseteq_{tr} \Theta$. By Theorem 3.3, O is linearizable. Further, because divergence-sensitive branching bisimulation implies next-free LTL formula [14], the lock-free property can be preserved by $\sim_{\mathcal{B}}^{div}$. By Lemma 5.3, Θ is a lock-free abstract object, so if $O \sim_{\mathcal{B}}^{div} \Theta$, then O is lock-free. The case of wait-free property can be proved similarly. $\square$

6. Experiments on Verifying Lock-Free Stacks

To show the efficiency and effectiveness of our method for proving linearizability and progress properties, we conduct the experiment on verifying non-blocking stacks [16, 27, 30]. We employ the Construction and Analysis of Distributed Processes (CADP) toolbox [20] to model and verify the algorithms.

6.1. Analyzing the concurrent stack with hazard pointers

Treiber stack. Figure 4 (Lines 1-21) shows the Treiber's stack algorithm [16], which involves two methods `push` and `pop`. The stack is implemented as a linked-list pointed by the `Top` pointer (shared variable). Both operations modify the stack by doing a `CAS`. The linearization point of `push` is Line 7 if the `CAS` succeeds, and the linearization point of `pop` is either Line 14 if the stack is empty; or Line 18 if the `CAS` succeeds.

Memory reclamation - hazard pointers. We further consider a complicate concurrent stack that involves with hazard pointers, which provides a memory reclamation mechanism to avoid the ABA problem [27]. The hazard pointer variation is shown in Figure 4 (Lines 22-45), which includes `push_HP` and `pop_HP`. Method `push_HP` is the same as `push`. Method `pop_HP` involves a hazard pointer before the ABA-prone code (Line 37), and calls the method `Retire_Node(o)` (Line 43) after the successful `CAS` operation to determine which retired locations can be freed. The codes of `Retire_Node(o)` can be found in [27]. These operations are used to reclaim memory blocks safely, which is not relevant to the effect of the method call. Further to achieve the desired lock-freedom of the stack, it is required that method `Retire_Node(o)` should be wait-free.

Figure 5 presents an abstract stack as the specification, where each method body of methods `push_spec` and `pop_spec` is implemented by an atomic block.

The linearization point of `push`/`push_HP` and `pop`/`pop_HP` of the concurrent stacks is independent of the dynamic execution, which can be simply located on the implementation code such that the effect of the method call always takes place at the static linearization point between the method call and the corresponding method return. Therefore, the *effect* of each mehtod call can be abstracted into the atomic operation of the linearizable specification; and vice versa, the atomic operation of the specification can be refined to several τ-steps including a linearization point of the concurrent stack. Such the corresponding relation can be naturally characterized by the branching bisimulation equivalence defined in Definition 4.1. Our experiment confirms the existence of the branching bisimulation equivalence relation between the concurrent stack (with hazard pointers) in Figure 4 and the abstract stack in Figure 5. To check the progress condition, the explicit divergence in branching bisimulation is also need to be considered.

```
class Node { int value;  Node next;  Node (int value) { value = value;  next = null; } }   Node Top;  // Top is a shared variable.
void init() { Top := null }
```

```
void push(v) {
bool done:=false;
Node x:=new_node(v);
while(!done) {
 Node old:=Top;
 x.next:=old;
 done:=CAS(&Top,old,x);
 }
return;
}
```

```
int pop() {
bool done:=false;
while(!done) {
Node old:=Top;
 if (old==null)
   then return EMPTY
 Node x:=old.next;
 done:=CAS(&Top,old,x);
 }
return old.data;
}
```

```
void push_HP(v) {
bool done:=false;
Node x:=new_node(v);
while(!done) {
 Node old:=Top;
 x.next:=old;
 done:=CAS(&Top,old,x);
 }
return;
}
```

```
pop_HP() {
done:=false;
while(!done) {
 Node old:=Top;
 if(old=null) return EMPTY;
 *hp:=old;
 if(Top==old)
   x:=old.next;
   done:=cas(&Top,old,x);
   }
v:=o.value;
Retire_Node(o);
return v;
}
```

Figure 4: Treiber's lock-free stack push/pop and the lock-free stack with hazard pointers push_HP/pop_HP, where the implementation of Retire_Node(o) can be found in [27].

#th/#op	Δ_{spec}	Δ_{Tr}	Verification time (in s) (Δ_{Tr})		Δ_{HP}	Verification time (in s) (Δ_{HP})	
			Thm 5.4 (linearizability/lock-free)	Thm 3.3 (linearizability)		Thm 5.4 (linearizability/lock-free)	Thm 3.3 (linearizability)
2/1	70	81	0.73	0.80	195	0.76	0.97
2/2	487	823	0.74	0.92	7493	0.80	2.93
2/3	1678	3673	0.81	2.15	93352	1.45	25.30
2/4	4237	10999	0.93	5.08	808079	6.67	232.02
2/5	8920	26101	1.04	11.04	5447816	101.11	10514.94
2/6	16651	53197	1.20	21.89	31953747	283.87	>16h
2/7	28516	97435	1.63	39.55	174455921	1528.88	>10h
2/8	45769	164881	2.14	63.14	M. O.	-	-
3/1	706	1036	2.25	2.92	8988	2.51	9.11
3/2	12341	40309	1.09	21.48	4937828	76.26	>10h
3/3	77850	411772	4.11	230.60	M. O.	-	-
3/4	314392	2247817	19.90	>10h	M. O.	-	-
4/1	6761	15595	0.71	11.36	612665	9.36	807.35
4/2	304197	2351919	22.52	4665.74	M. O.	-	-
5/1	64351	261527	10.48	816	60598453	408.27	>10h
6/1	616838	4771785	141.86	81316.58	M. O.	-	-

Table 1: Experimental results on verifying the lock-free stacks with/without hazard pointers.

```
void push_spec(int x)
atomic {
Node node:=new Node(x);
node.next:=Top;
Top:=node;
}
return;

int pop_spec()
atomic {
if (Top == null) b:=false
else
Node Curr_Top:=Top;
Top:=Curr_Top.next; b:=true
}
if (b==false)
then return EMPTY
else return Curr_Top.data
```

Figure 5: The linearizable specification of concurrent stacks.

6.2. Experimental results

To verify linearizability and lock-free property of concurrent stacks, we check that the abstract object in Figure 5 and the two concrete objects in Figure 4 are divergence-sensitive branching bisimilar. For the automated verification, we conduct the experiment on finite systems. All experiments run on a server which is equipped with a 4×24-core Intel CPU @ 2.3 GHz and 1024 GB memory under 64-bit Centos 7.2.

The verification results of both versions of the concurrent stacks are shown in Table 1, where M.O. means memory out. The first column indicates the number of threads and operations. The second and third columns indicate the state space of the specification Δ_{spec} and Treiber's stack Δ_{Tr} respectively; the fourth and the fifth columns indicates the verification time (in seconds) of Δ_{Tr} based on bisimulation technique (Theorem 5.4) and trace refinement checking (Theorem 3.3). The next column shows the state space of the variant stack with hazard pointers Δ_{HP}. And the last two columns indicate the verification time (in seconds) of Δ_{HP} based on the two kinds of methods.

From the experimental results, we can see that, the branching bisimulation method is much more feasible and efficient than trace refinement checking for finite-state systems. For example, based on the bisimulation equivalence checking, the verification of Δ_{HP} with about 5 million states (in the case 2/5) takes around 100 seconds, while trace refinement checking takes around 3 hours. For the most million states, the time of trace refinement checking is greater than 10 hours.

6.3. Bugs hunting - the stack with revised hazard pointers [30]

While we verify the correctness of the concurrent stack with the revised hazard pointers in [30], a counter-example (a trace which witnesses the concrete object and the specification are not divergence-sensitive branching bisimilar) is given by CADP with just two concurrent threads. The revised version of the hazard pointers shown in Figure 7 avoids the ABA problem at the expense of violating the wait-free property of hazard pointers in the original algorithm [27]. The error-path ends in a self-loop (at state 20 in Figure 6) in which one thread keeps reading the same hazard pointer value of another thread in function `retireNode(t)` without making any progress and causes the divergent path of the stack. So the revised stack in [30] is not lock-free.

```
initial state 0
"CALL !PUSH !2"
state 44
"i"
state 2
"RET !PUSH !2"
state 41
"CALL !PUSH !2"
state 262
"CALL !POP !1"
state 313
"i"
state 312
"i"
state 187
"i"
state 186
"RET !PUSH !2"
state 71
"CALL !POP !2"
state 20
```

Figure 6: The counter-example of trace generated by CADP.

```
pop() {
local done,next,t,t1;
done:=false;
while(!done) {
t:=Top;
if(t==null)
  return EMPTY;
HP[tid]:=t;
t1:=Top;
if(t==t1) {
 next:=t.Next;
 done:=cas(&Top,t,next) }
}
retireNode(t);
HP[tid]:=null;
return t;
}
```

```
retireNode(t);
local i,t';
i:=1;
while(i<=th_num) {
if(i!=tid) {
  t':=HP[i];
  if (t'!=t) {
   i:=i+1;}
  }
else i:=1+1;
}
}
```

Figure 7: A buggy implementation of the lock-free stack [30].

Summary. Our experiment presents the following advantages: (i) We provide a *uniform* notion $\sim_{\mathcal{B}}^{div}$ to verify both the linearizability and lock-freedom of the variant of concurrent stacks; (ii) Our techniques are fully automated (for finite-state systems) and rely on the efficient existing branching bisimulation algorithms. (iii) In contrast to proof techniques [3, 5, 28] for linearizabilty and progress, our method is able to generate counterexamples in an automated manner.

Our method is built on the behaviour of objects, not on their statically syntactic constructs. Thus, for any concurrent objects, we do not care how the object algorithm is constructed, but only concerns state transition relations between the abstract and concrete system models. Therefore, our method can be applied to general concurrent object programs.

Finally, we like to point out that divergence-sensitive branching bisimulation implying $\mathrm{LTL}_{-\bigcirc}$ equivalence applies to any countable transition system. This implies that all progress properties of abstract object programs are carried over to bisimilar concrete ones. For analysing infinite systems, branching bisimulation can be done using standard proof techniques such as parameterised boolean equation systems [26]. For finite-state systems, polynomial-time algorithms can be employed [25]. Note that in the latter setting, trace refinement is PSPACE-complete.

7. Related Work and Conclusion

As we mentioned in the introduction, almost all the work in the literature on the verification of the correctness of concurrent objects are based on the refinement techniques. Our work takes the first step to explore the divergence-sensitive bisimulation equivalence technique to verify linearizability and progress properties of concurrent objects [1].

Our transition system models are closely related to [8]. The key idea of [8] is to construct a linearizable specification using labeled transition systems (LTSs) and describe linearizability as the trace refinement relation between the specification and the concurrent algorithm. They also use abstract objects and state transition systems as semantic models for describing the behaviour of concurrent objects, which correspond to our abstract and concrete object systems. However, the trace refinement relation they proposed is only suitable for checking linearizability, not for checking other properties e.g., progress properties. Moreover, they do not discuss the (bi)-simulation relations between an abstract program and a concrete object program.

In our work, we reveal that there exists a natural (branching) bisimulation equivalence between the abstract and the concrete object if we use a coarse-grained concurrent object as the specification. Moreover, when do model checking, for finite-state systems, polynomial-time branching bisimulation checking algorithms is provided [25] - in contrast to (classical) PSPACE-complete trace refinement - for checking linearizability.

Different from [8], where the trace refinement is shown to be equivalent to linearizability, our work (Theorem 4.2, Theorem 5.4) shows that (divergence-sensitive) branching bisimulation implies linearizability. If an object system and the specification are (divergence-sensitive) branching bisimilar, then the object is linearizable; but if they are not (divergence-sensitive) branching bisimilar, then it will generate a counterexample, which needs to be analyzed to determine whether it violates the linearizability (e.g., it may violate the lock-free property).

Other model checking methods on verifying linearizability are [21, 22, 23]. Cerny et al. [21] propose an method automata

[1] This paper is a revised version of our technical report [9].

to verify linearizability of concurrent linked-list implementations, but only for the execution of two fixed operations. Vechev et al. [22] use SPIN model checker to model and verify linearizability. Burckhardt et al. [23] present an automatic linearizability checker Line-Up based on the model checker CHESS.

To verify progress properties, Gotsman etal. [7] first propose a generalisation of linearizability such that it can specify liveness. Based on the definition, they reveal a connection between lock-free property and a termination-sensitive contextual refinement, but they do not discuss other progress properties. Liang etal. [6, 28] study five progress properties and show that each progress property is equivalent to a specific termination-sensitive contextual refinement. But they need to define different refinement relations for different progress properties. In our work, divergence-sensitive branching bisimulation implies $LTL_{-\bigcirc}$ equivalence and can preserve all progress properties that are specified in $LTL_{-\bigcirc}$. So there is no need to define different relations for different progress properties. Further, our recent work [29] extends the proposed bisimulation method such that it can verify more complicated concurrent objects as in [5, 28].

Conclusion. This paper proposes a novel and efficient method based on divergence-sensitive branching bisimulation for verifying linearizability and lock-free property of concurrent objects. Instead of using trace refinement to check linearizability, our approach enables the use of branching bisimulation techniques for checking both the linearizability and progress properties of concurrent objects. Our experiment presents that (divergence-sensitive) branching bisimulation is an inherent equivalence relation between the Treiber's lock-free stack (and the revised stack with hazard pointers) and the corresponding concurrent specification. Our method also reveals a new bug violating the lock-freedom.

References

[1] M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. ACM Trans. on Program. Lang. and Syst. 12(3): 463-492. 1990.

[2] S. Owicki and L.Lamport. Proving Liveness Properties of Concurrent Programs. ACM Transactions on Programming Languages and Systems. Vol.4(3):455-495. 1982.

[3] Vafeiadis Victor. Modular Fine-Grained Concurrency Verification. Cambridge University, Phd Thesis. 2008.

[4] H. Liang, X. Feng, M. Fu. A Rely-Guarantee-based Simulation for Verifying Concurrent Program Transformations. In: POPL:455-468. 2012.

[5] H. Liang, X. Feng. Modular Verification of Linearizability with Non-Fixed Linearization Points. In: PLDI, pages 459-470. 2013.

[6] H. Liang, J. Hoffmann, X. Feng, Z. Shao. Characterizing Progress Properties of Concurrent Objects via Contextual Refinements. In: CONCUR, LNCS 8052, pages: 227-241. 2013.

[7] Alexey Gotsman, Hongseok Yang. Liveness-Perseving Atomicity Abstraction. In: ICALP, LNCS:6756: 453-465. 2011.

[8] Y. Liu, W. Chen, *et al.* Verifying Linearizabiity via Optmized Refinement Checking. IEEE Trans. on Soft. Eng. 39(7): 1018-1039. 2013.

[9] X. Yang and J. -P. Katoen. Proving Linearizability and Progress of Concurrent Objects by Bisimulation. Technical Report ISCAS-SKLCS-14-16. Sep. 2014.

[10] P. Cerny, A. Radhakrishna, *et al.* Model Checking of Linearizability of Concurrent List Implementations. In: CAV, pages 465-479. 2010.

[11] I. Filipovic, P. O'Hearn, N. Rinetzky, H. Yang. Abstraction for Concurrent Objects. Theoretical Computer Science. Vol. 411(51-52): 4379-4398. 2010.

[12] M. Herlihy, N. Shavit. The Art of Multiprocessor Programming. Morgan Kaufmann, April, 2008.

[13] D. Hendler, N. Shavit, L. Yerushalmi. A Scalable Lock-Free Stack Algorithm. In: SPAA, pages 206-215. 2004.

[14] C. Baier and J.-P. Katoen. Principles of Model Checking. The MIT Press. 2008.

[15] E. Petrank, M. Musuvathi, *et al.* Progress Guarantee for Parallel Programs via Bounded Lock-Freedom. In: PLDI, pages 144-154. 2009.

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

[17] Rob J. van Glabbeek and W. Peter Weijland. Branching Time and Abstraction in Bisimulation Semantics. Journal of the ACM. Vol.43(3):555-600. 1996.

[18] M. C. Browne, E. M. Clarke and O. Grumberg. Characterizing Finite Kripke Structures in Propositional Temporal Logic. Theoretical Computer Science. Vol.59(1-2):115-131. 1988.

[19] Kedar S. Namjoshi. A Simple Characterization of Stuttering Bisimulation. In: FSTTCS, LNCS 1346, pages 284-296. 1997.

[20] Radu Mateescu and Wendelin Serwe. Model Checking and Performance Evaluation with CADP Illustrated on Shared-Memory Mutual Exclusion Protocols. Vol.78(7):843-861. Science of Computer Programming. 2013.

[21] R. Colvin, L. Groves, V. Luchangco and M. Moir. Formal Verification of a Lazy Concurrent List-Based Set Algorithm. In: CAV, LNCS 4144, pages 475-488. 2006.

[22] M. T. Vechev, E. Yahav, and G. Yorsh. Experience with model checking linearizability. In: SPIN, pages: 261-278. 2009.

[23] S. Burckhardt, *et al*. Line-Up: A Complete and Automatic Linearizability Checker. In: PLDI, pages 330-340. 2010.

[24] B. Dongol. Formalising Progress Properties of Non-Blocking Programs. In: ICFEM, pages 284-303. 2006.

[25] J. F. Groote, F. Vaandrager. An Efficient Algorithm for Branching Bisimulation and Stutter Equivalence. In: Automata, Language and Programming, LNCS 443, pages 626-638. 1990.

[26] T. Chen, B. Ploeger, J. van de Pol and T. Willemse. Equivalence Checking for Infinite Systems Using Parameterized Boolean Equation Systems. In: CONCUR, vol.4703, pages: 120-135. 2007.

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

[28] H. Liang, X. Feng, Z. Shao. Compositional verification of termination-preserving refinement of concurrent programs. CSL-LICS, 2014: 65:1-65:10.

[29] X. Yang, J. -P. Katoen, H. Lin and H. Wu. Proving Linearizability via Branching Bisimulation. CoRR abs/1609.07546. Sep., 2016.

[30] M. Fu, Y. Li, X. Feng, Z. Shao, and Y. Zhang. Reasoning about Optimistic Concurrency Using a Program Logic for History. In *CONCUR 2010*, LNCS vol.6269, pages 388–402. Springer, 2010.