Characterizing Progress Properties of Concurrent Objects via Contextual Refinements

Hongjin Liang[1,2], Jan Hoffmann[2], Xinyu Feng[1], and Zhong Shao[2]

[1] University of Science and Technology of China
[2] Yale University

Abstract. Implementations of concurrent objects should guarantee linearizability and a progress property such as wait-freedom, lock-freedom, obstruction-freedom, starvation-freedom, or deadlock-freedom. Conventional informal or semi-formal definitions of these progress properties describe conditions under which a method call is guaranteed to complete, but it is unclear how these definitions can be utilized to formally verify system software in a layered and modular way.

In this paper, we propose a unified framework based on contextual refinements to show exactly how progress properties affect the behaviors of client programs. We give formal operational definitions of all common progress properties and prove that for linearizable objects, each progress property is equivalent to a specific type of contextual refinement that preserves termination. The equivalence ensures that verification of such a contextual refinement for a concurrent object guarantees both linearizability and the corresponding progress property. Contextual refinement also enables us to verify safety and liveness properties of client programs at a high abstraction level by soundly replacing concrete method implementations with abstract atomic operations.

1 Introduction

A concurrent object consists of shared data and a set of methods that provide an interface for client threads to manipulate and access the shared data. The synchronization of simultaneous data access within the object affects the progress of the execution of the client threads in the system.

Various progress properties have been proposed for concurrent objects. The most important ones are wait-freedom, lock-freedom and obstruction-freedom for non-blocking implementations, and starvation-freedom and deadlock-freedom for lock-based implementations. These properties describe conditions under which method calls are guaranteed to successfully complete in an execution. For example, lock-freedom guarantees that "infinitely often some method call finishes in a finite number of steps" [9].

Nevertheless, the common informal or semi-formal definitions of the progress properties are difficult to use in a modular and layered program verification because they fail to describe how the progress properties affect clients. In a modular

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 227–241, 2013.

verification of client threads, the concrete implementation Π of the object methods should be replaced by an abstraction (or specification) Π_A that consists of equivalent atomic methods. The progress properties should then characterize whether and how the behaviors of a client program will be affected if a client uses Π instead of Π_A. In particular, we are interested in systematically studying whether the termination of a client using the abstract methods Π_A will be preserved when using an implementation Π with some progress guarantee.

Previous work on verifying the *safety* of concurrent objects (*e.g.*, [4,12]) has shown that linearizability—a standard safety criterion for concurrent objects—and contextual refinement are equivalent. Informally, an implementation Π is a contextual refinement of a (more abstract) implementation Π_A, if every observable behavior of any client program using Π can also be observed when the client uses Π_A instead. To obtain equivalence to linearizability, the observable behaviors include I/O events but not divergence (*i.e.*, non-termination). Recently, Gotsman and Yang [6] showed that a client program that diverges using a linearizable and *lock-free* object must also diverge when using the abstract operations instead. Their work reveals a connection between lock-freedom and a form of contextual refinement which preserves termination as well as safety properties. It is unclear how other progress guarantees affect termination of client programs and how they are related to contextual refinements.

This paper studies all five commonly used progress properties and their relationships to contextual refinements. We propose a unified framework in which a certain type of termination-sensitive contextual refinement is equivalent to linearizability together with one of the progress properties. The idea is to identify different observable behaviors for different progress properties. For example, for the contextual refinement for lock-freedom we observe the divergence of the whole program, while for wait-freedom we also need to observe which threads in the program diverge. For lock-based progress properties, *e.g.*, starvation-freedom and deadlock-freedom, we have to take fair schedulers into account.

Our paper makes the following new contributions:

- We formalize the definitions of the five most common progress properties: wait-freedom, lock-freedom, obstruction-freedom, starvation-freedom, and deadlock-freedom. Our formulation is based on possibly infinite event traces that are operationally generated by any client using the object.
- Based on our formalization, we prove relationships between the progress properties. For example, wait-freedom implies lock-freedom and starvation-freedom implies deadlock-freedom. These relationships form a lattice shown in Figure 1 (where the arrows represent implications). We close the lattice with a bottom element that we call *sequential termination*, a progress property in the sequential setting. It is weaker than any other progress property.
- We develop a unified framework to characterize progress properties via contextual refinements. With linearizability, each progress property is proved equivalent to a contextual refinement which takes into account divergence of programs. A companion TR [14] contains the formal proofs of our results.

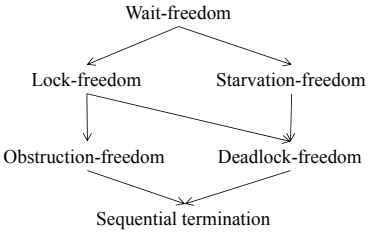

Fig. 1. Relationships between Progress Properties

By extending earlier equivalence results on linearizability [4], our contextual refinement framework can serve as a new alternative definition for the full correctness properties of concurrent objects. The contextual refinement implied by linearizability and a progress guarantee precisely characterizes the properties at the abstract level that are preserved by the object implementation. When proving these properties of a client of the object, we can soundly replace the concrete method implementations by its abstract operations. On the other hand, since the contextual refinement also implies linearizability and the progress property, we can potentially borrow ideas from existing proof methods for contextual refinements, such as simulations (*e.g.*, [13]) and logical relations (*e.g.*, [2]), to verify linearizability and the progress guarantee together.

In the remainder of this paper, we first informally explain our framework in Section 2. We then introduce the formal setting in Section 3; including the definition of linearizability as the safety criterion of objects. We formulate the progress properties in Section 4 and the contextual refinement framework in Section 5. We discuss related work and conclude in Section 6.

2 Informal Account

In this section, we informally describe our results. We first give an overview of linearizability and its equivalence to the basic contextual refinement. Then we explain the progress properties and summarize our new equivalence results.

Linearizability and Contextual Refinement. *Linearizability* is a standard safety criterion for concurrent objects [9]. Intuitively, linearizability describes atomic behaviors of object implementations. It requires that each method call should appear to take effect instantaneously at some moment between its invocation and return.

Linearizability intuitively establishes a correspondence between the object implementation Π and the intended atomic operations Π_A. This correspondence can also be understood as a *contextual refinement*. Informally, we say that Π is a contextual refinement of Π_A, $\Pi \sqsubseteq \Pi_A$, if substituting Π for Π_A in any context (*i.e.*, in a client program) does not add observable behaviors. External observers cannot tell that Π_A has been replaced by Π from monitoring the behaviors of the client program.

It has been proved [4,12] that linearizability is equivalent to a contextual refinement in which the observable behaviors are finite traces of I/O events. Thus this basic contextual refinement can be used to distinguish linearizable objects from non-linearizable ones. But it cannot characterize progress properties of objects.

Progress Properties. Figure 2 shows several implementations of a counter with different progress guarantees that we study in this paper. A counter object provides the two methods `inc` and `dec` for incrementing and decrementing a shared variable `x`. The implementations given here are not intended to be practical but merely to demonstrate the meanings of the progress properties. We assume that every command is executed atomically.

Informally, an object implementation is *wait-free*, if it guarantees that every thread can complete any started operation of the data structure in a finite number of steps [7]. Figure 2(a) shows an ideal wait-free implementation in which the increment and the decrement are done atomically. This implementation is obviously wait-free since it guarantees termination of every method call regardless of interference from other threads. Note that realistic implementations of wait-free counters are more complex and involve arrays and atomic snapshots [1].

Lock-freedom is similar to wait-freedom but only guarantees that *some* thread will complete an operation in a finite number of steps [7]. Typical lock-free implementations (such as the well-known Treiber stack, HSY elimination-backoff stack and Harris-Michael lock-free list) use the atomic compare-and-swap instruction `cas` in a loop to repeatedly attempt an update until it succeeds. Figure 2(b) shows such an implementation of the counter object. It is lock-free, because whenever `inc` and `dec` operations are executed concurrently, there always exists some successful update. Note that this object is not wait-free. For the following program (2.1), the `cas` instruction in the method called by the left thread may continuously fail due to the continuous updates of `x` made by the right thread.

```
inc();   ||   while(true) inc();                    (2.1)
```

Herlihy *et al.* [8] propose *obstruction-freedom* which "guarantees progress for any thread that eventually executes in isolation" (*i.e.*, without other active threads in the system). They present two double-ended queues as examples. In Figure 2(c) we show an obstruction-free counter that may look contrived but nevertheless illustrates the idea of the progress property.

The implementation introduces a variable `i`, and lets `inc` perform the atomic increment after increasing `i` to 10 and `dec` do the atomic decrement after decreasing `i` to 0. Whenever a method is executed in isolation (*i.e.*, without interference from other threads), it will complete. Thus the object is obstruction-free. It is not lock-free, because for the client

```
inc();   ||   dec();                               (2.2)
```

which executes an increment and a decrement concurrently, it is possible that neither of the method calls returns. For instance, under a specific schedule, every increment over `i` made by the left thread is immediately followed by a decrement from the right thread.

```
1 inc() { x := x + 1; }
2 dec() { x := x - 1; }
```

(a) Wait-Free (Ideal) Impl.

```
1 inc() {
2  local t, b;
3  do {
4    t := x;
5    b := cas(&x,t,t+1);
6  } while(!b);
7 }
```

(b) Lock-Free Impl.

```
 1 inc() {
 2   while (i < 10) {
 3     i := i + 1;
 4   }
 5   x := x + 1;
 6 }
 7 dec() {
 8   while (i > 0) {
 9     i := i - 1;
10   }
11   x := x - 1;
12 }
```

(c) Obstruction-Free Impl.

```
1 inc() {
2   TestAndSet_lock();
3   x := x + 1;
4   TestAndSet_unlock();
5 }
```

(d) Deadlock-Free Impl.

```
 1  inc() {
 2    Bakery_lock();
 3    x := x + 1;
 4    Bakery_unlock();
 5  }
```

(e) Starvation-Free Impl.

Fig. 2. Counter Objects with Methods `inc` and `dec`

Wait-freedom, lock-freedom, and obstruction-freedom are progress properties for non-blocking implementations, where a delay of a thread cannot prevent other threads from making progress. In contrast, deadlock-freedom and starvation-freedom are progress properties for lock-based implementations. A delay of a thread holding a lock will block other threads which request the lock.

Deadlock-freedom and starvation-freedom are often defined in terms of locks and critical sections. Deadlock-freedom guarantees that some thread will succeed in acquiring the lock, and starvation-freedom states that every thread attempting to acquire the lock will eventually succeed [9]. For example, a test-and-set spin lock is deadlock-free but not starvation-free. In a concurrent access, some thread will successfully set the bit and get the lock, but there might be a thread that is continuously failing to get the lock. Lamport's bakery lock is starvation-free. It ensures that threads can acquire locks in the order of their requests.

However, as noted by Herlihy and Shavit [10], the above definitions based on locks are unsatisfactory, because it is often difficult to identify a particular field in the object as a lock. Instead, they suggest defining them in terms of method calls. They also notice that the above definitions implicitly assume that every thread acquiring the lock will eventually release it. This assumption requires *fair* scheduling, *i.e.*, every thread gets eventually executed.

Following Herlihy and Shavit [10], we say an object is *deadlock-free*, if in each *fair* execution there always exists some method call that can finish. As an example in Figure 2(d), we use a test-and-set lock to synchronize the increments of the counter. Since some thread is guaranteed to acquire the test-and-set lock, the method call of that thread is guaranteed to finish. Thus the object is deadlock-free. Similarly, a *starvation-free* object guarantees that every method call can finish in fair executions. Figure 2(e) shows a counter implemented with Lamport's bakery lock. It is starvation-free since the bakery lock ensures that every thread can acquire the lock and hence every method call can eventually complete.

Table 1. Characterizing Progress Properties via Contextual Refinements $\Pi \sqsubseteq \Pi_A$

	Wait-Free	Lock-Free	Obstruction-Free	Deadlock-Free	Starvation-Free
Π_A	(t, Div.)	Div.	Div.	Div.	(t, Div.)
Π	(t, Div.)	Div.	Div. if Isolating	Div. if Fair	(t, Div.) if Fair

Our Results. None of the above definitions of the five progress properties describes their guarantees regarding the behaviors of client code. In this paper, we define several contextual refinements to characterize the effects over client behaviors when the client uses objects with some progress properties. We show that linearizability together with a progress property is equivalent to a certain termination-sensitive contextual refinement. Table 1 summarizes our results.

For each progress property, the new contextual refinement $\Pi \sqsubseteq \Pi_A$ is defined with respect to a divergence behavior and/or a specific scheduling at the implementation level (the third row in Table 1) and at the abstract side (the second row), in addition to the I/O events in the basic contextual refinement for linearizability.

- For wait-freedom, we need to observe the divergence of each individual thread t, represented by "(t, Div.)" in Table 1, at both the concrete and the abstract levels. We show that, if the thread t of a client program diverges when the client uses a linearizable and wait-free object Π, then thread t must also diverge when using Π_A instead.
- The case for lock-freedom is similar, except that we now consider the divergence behaviors of the *whole* client program rather than individual threads (denoted by "Div." in Table 1). If a client diverges when using a linearizable and lock-free object Π, it must also diverge when it uses Π_A instead.
- For obstruction-freedom, we consider the behaviors of *isolating* executions at the concrete side (denoted by "Div. if Isolating" in Table 1). In those executions, eventually only one thread is running. We show that, if a client diverges in an isolating execution when it uses a linearizable and obstruction-free object Π, it must also diverge in some abstract execution.
- For deadlock-freedom, we only care about *fair* executions at the concrete level (denoted by "Div. if Fair" in Table 1).
- For starvation-freedom, we observe the divergence of each individual thread at both levels and restrict our considerations to fair executions for the concrete side ("(t, Div.) if Fair" in Table 1). Any thread using Π can diverge in a fair execution, only if it also diverges in some abstract execution.

These new contextual refinements can characterize linearizable objects with progress properties. We will formalize the results and give examples in Section 5.

3 Formal Setting and Linearizability

In this section, we formalize linearizability and show its equivalence to a contextual refinement that preserves safety properties only. This equivalence is the basis of our new results that relate progress properties and contextual refinements.

$$
\begin{array}{rrcl}
(\mathit{Expr}) & E & ::= & \ldots \quad (\mathit{BExp})\ B ::= \ldots \quad (\mathit{Instr})\ c ::= \mathbf{print}(E) \mid \ldots \\
(\mathit{Stmt}) & C & ::= & \mathbf{skip} \mid c \mid x := f(E) \mid \mathbf{return}\ E \mid \mathbf{end} \\
 & & & \mid \langle C \rangle \mid C; C \mid \mathbf{if}\ (B)\ C\ \mathbf{else}\ C \mid \mathbf{while}\ (B)\{C\} \\
(\mathit{Prog}) & W & ::= & \mathbf{skip} \mid \mathbf{let}\ \Pi\ \mathbf{in}\ C \| \ldots \| C \\
(\mathit{ODecl}) & \Pi & ::= & \{f_1 \rightsquigarrow (x_1, C_1), \ldots, f_n \rightsquigarrow (x_n, C_n)\}
\end{array}
$$

Fig. 3. Syntax of the Programming Language

$$
\begin{array}{rrcl}
(\mathit{State}) & \mathcal{S} & ::= & \ldots \qquad (\mathit{ThrdID})\ \mathsf{t} \in \mathit{Nat} \\
(\mathit{Evt}) & e & ::= & (\mathsf{t}, f, n) \mid (\mathsf{t}, \mathbf{ret}, n) \mid (\mathsf{t}, \mathbf{obj}) \mid (\mathsf{t}, \mathbf{obj}, \mathbf{abort}) \\
 & & & \mid (\mathsf{t}, \mathbf{out}, n) \mid (\mathsf{t}, \mathbf{clt}) \mid (\mathsf{t}, \mathbf{clt}, \mathbf{abort}) \mid (\mathsf{t}, \mathbf{term}) \mid (\mathbf{spawn}, n) \\
(\mathit{ETrace}) & T & ::= & \epsilon \mid e :: T \quad \text{(co-inductive)}
\end{array}
$$

Fig. 4. States and Event Traces

Language and Semantics. We use a similar language as in previous work of Liang and Feng [12]. As shown in Figure 3, a program W consists of several client threads that run in parallel. Each thread could call the methods declared in the object Π. A method f is defined as a pair (x, C), where x is the formal argument and C is the method body. The object Π could be either concrete with fine-grained code that we want to verify, or abstract (usually denoted as Π_A in the following) that we consider as the specification. For the latter case, each method body should be an atomic operation of the form $\langle C \rangle$ and it should be always safe to execute it. For simplicity, we assume there is only one object in the program W and each method takes one argument only.

Most commands are standard. Clients can use $\mathbf{print}(E)$ to produce observable external events. We do not allow the object's methods to produce external events. To simplify the semantics, we also assume there are no nested method calls. To discuss progress properties later, we introduce an auxiliary command **end**. It is a special marker that can be added at the end of a thread, but is not supposed to be used directly by programmers. The **skip** statement plays two roles here: a statement that has no computation effects or a flag to show the end of an execution.

We use $\mathcal{S}$ for a program state. Program transitions $(W, \mathcal{S}) \overset{e}{\longmapsto} (W', \mathcal{S}')$ generate events e defined in Figure 4. A method invocation event (t, f, n) is produced when thread t executes $x := f(E)$, where n is the value of the argument E. A return $(\mathsf{t}, \mathbf{ret}, n)$ is produced with the return value n. $\mathbf{print}(E)$ generates an output $(\mathsf{t}, \mathbf{out}, n)$, and **end** generates a termination marker $(\mathsf{t}, \mathbf{term})$. Other steps generate either normal object actions $(\mathsf{t}, \mathbf{obj})$ (for steps inside method calls) or silent client actions $(\mathsf{t}, \mathbf{clt})$ (for client steps other than $\mathbf{print}(E)$). For transitions leading to the error state **abort** (*e.g.*, invalid memory access), fault events are produced: $(\mathsf{t}, \mathbf{obj}, \mathbf{abort})$ by the object method code and $(\mathsf{t}, \mathbf{clt}, \mathbf{abort})$ by the client code. We also introduce an auxiliary event $(\mathbf{spawn}, n)$, saying that n threads are spawned. It will be useful later when defining fair scheduling (in Section 4). We write $\mathsf{tid}(e)$ for the thread ID in the event e. The predicate $\mathsf{is\_clt}(e)$

$$\mathcal{T}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{T \mid \exists W', \mathcal{S}'.\ (W,\mathcal{S}) \stackrel{T}{\longmapsto}{}^{*} (W',\mathcal{S}') \vee (W,\mathcal{S}) \stackrel{T}{\longmapsto}{}^{*} \mathbf{abort}\}$$
$$\mathcal{H}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_hist}(T) \mid T \in \mathcal{T}[\![W,\mathcal{S}]\!]\}$$
$$\mathcal{O}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}[\![W,\mathcal{S}]\!]\}$$

Fig. 5. Generation of Finite Event Traces

states that the event e is either a silent client action, an output, or a client fault. We write $\mathsf{is\_inv}(e)$ and $\mathsf{is\_ret}(e)$ to denote that e is a method invocation and a return, respectively. The predicate $\mathsf{is\_abt}(e)$ denotes a fault of the object or the client. Method invocations, returns and object faults are called *history* events, which will be used to define linearizability below. Outputs, client faults and object faults are called *observable* events.

An event trace T is a finite or infinite sequence of events. We write $T(i)$ for the i-th event of T. $\mathsf{last}(T)$ is the last event in a finite T. The trace $T(1..i)$ is the sub-trace $T(1), \ldots, T(i)$ of T, and $|T|$ is the length of T ($|T| = \omega$ if T is infinite). The trace $T|_{\mathsf{t}}$ represents the sub-trace of T consisting of all events whose thread ID is t. We can use $\mathsf{get\_hist}(T)$ to project T to the sub-trace consisting of all the history events, and $\mathsf{get\_obsv}(T)$ for the sub-trace of all the observable events. Finite traces of history events are called *histories*.

In Figure 5, we define $\mathcal{T}[\![W,\mathcal{S}]\!]$ for the prefix-closed set of finite traces produced by the executions of $(W,\mathcal{S})$. We use $(W,\mathcal{S}) \stackrel{T}{\longmapsto}{}^{*} (W',\mathcal{S}')$ for zero or multiple-step program transitions that generate the trace T. We also define $\mathcal{H}[\![W,\mathcal{S}]\!]$ and $\mathcal{O}[\![W,\mathcal{S}]\!]$ to get histories and finite observable traces produced by the executions of $(W,\mathcal{S})$. The TR [14] contains more details about the language.

Linearizability and Basic Contextual Refinement. We formulate linearizability following its standard definition [11]. Below we sketch the basic concepts. Detailed formal definitions can be found in the companion TR [14].

Linearizability is defined using histories. We say a return e_2 *matches* an invocation e_1, denoted as $\mathsf{match}(e_1, e_2)$, iff they have the same thread ID. An invocation is *pending* in T if no matching return follows it. We can use $\mathsf{pend\_inv}(T)$ to get the set of pending invocations in T. We handle pending invocations in a history T in the standard way [11]: we append zero or more return events to T, and drop the remaining pending invocations. The result is denoted by $\mathsf{completions}(T)$. It is a set of histories, and for each history in it, every invocation has a matching return event.

Definition 1 (Linearizable Histories). $T \preceq_{\mathsf{lin}} T'$ *iff*

1. $\forall \mathsf{t}.\ T|_{\mathsf{t}} = T'|_{\mathsf{t}}$;
2. *there exists a bijection* $\pi : \{1, \ldots, |T|\} \to \{1, \ldots, |T'|\}$ *such that* $\forall i.\ T(i) = T'(\pi(i))$ *and* $\forall i, j.\ i < j \wedge \mathsf{is\_ret}(T(i)) \wedge \mathsf{is\_inv}(T(j)) \implies \pi(i) < \pi(j)$.

That is, T is linearizable *w.r.t.* T' if the latter is a permutation of the former, preserving the order of events in the same threads and the order of the non-overlapping method calls. Then an *object* is linearizable iff each of its concurrent histories after completions is linearizable *w.r.t.* some *legal sequential* history.

We use $\Pi_A \rhd (\mathcal{S}_a, T')$ to mean that T' is a legal sequential history generated by any client using the specification Π_A with an abstract initial state $\mathcal{S}_a$.

Definition 2 (Linearizability of Objects). *The object's implementation Π is linearizable w.r.t. Π_A under a refinement mapping φ, denoted by $\Pi \preceq_\varphi \Pi_A$, iff*

$$\forall n, C_1, \ldots, C_n, \mathcal{S}, \mathcal{S}_a, T.\ T \in \mathcal{H}[\![(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), \mathcal{S}]\!] \wedge (\varphi(\mathcal{S}) = \mathcal{S}_a)$$
$$\Longrightarrow \exists T_c, T'.\ T_c \in \mathsf{completions}(T) \wedge \Pi_A \rhd (\mathcal{S}_a, T') \wedge T_c \preceq_{\mathsf{lin}} T'\,.$$

Here the partial mapping φ: State $\rightharpoonup$ State relates concrete states to abstract ones.

The side condition $\varphi(\mathcal{S}) = \mathcal{S}_a$ in the above definition requires the initial concrete state $\mathcal{S}$ to be well-formed in that it represents a valid abstract state $\mathcal{S}_a$. For instance, φ may need $\mathcal{S}$ to contain a linked list and relate it to an abstract mathematical set in $\mathcal{S}_a$ for a set object. Besides, φ should always require the client states in $\mathcal{S}$ and $\mathcal{S}_a$ to be identical.

Next we define a contextual refinement between the concrete object and its specification, which is equivalent to linearizability.

Definition 3 (Basic Contextual Refinement). $\Pi \sqsubseteq_\varphi \Pi_A$ *iff*

$$\forall n, C_1, \ldots, C_n, \mathcal{S}, \mathcal{S}_a.\ (\varphi(\mathcal{S}) = \mathcal{S}_a)$$
$$\Longrightarrow \mathcal{O}[\![(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), \mathcal{S}]\!] \subseteq \mathcal{O}[\![(\textbf{let } \Pi_A \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), \mathcal{S}_a]\!]\,.$$

Remember that $\mathcal{O}[\![W, \mathcal{S}]\!]$ represents the prefix-closed set of observable event traces generated during the executions of $(W, \mathcal{S})$, which is defined in Figure 5.

Following Filipović *et al.* [4], we can prove that linearizability is equivalent to this contextual refinement. We give the proofs in the TR [14].

Theorem 4 (Basic Equivalence). $\Pi \preceq_\varphi \Pi_A \iff \Pi \sqsubseteq_\varphi \Pi_A$.

Theorem 4 allows us to use $\Pi \sqsubseteq_\varphi \Pi_A$ to identify linearizable objects. However, we cannot use it to characterize progress properties of objects. For the following example, $\Pi \sqsubseteq_\varphi \Pi_A$ holds although no concrete method call of `f` could finish (we assume this object contains a method `f` only).

Π(`f`) : `while(true) skip;` $\quad$ Π_A(`f`) : `skip;` $\quad$ C : `print(1); f(); print(1);`

The reason is that $\Pi \sqsubseteq_\varphi \Pi_A$ considers a *prefix-closed* set of event traces at the abstract side. For the above client C, the observable behaviors of **let** Π **in** C can all be found in the prefix-closed set of behaviors produced by **let** Π_A **in** C.

4 Formalizing Progress Properties

We define progress in Figure 6 as properties over both event traces T and object implementations Π. We say an object implementation Π has a progress property P iff all its event traces have the property. Here we use $\mathcal{T}_\omega$ to generate the event traces. Its definition in Figure 6 is similar to $\mathcal{T}[\![W, \mathcal{S}]\!]$ of Figure 5, but $\mathcal{T}_\omega[\![W, \mathcal{S}]\!]$ is for the set of finite or infinite event traces produced by *complete* executions. We use $(W, \mathcal{S}) \xrightarrow{T}{}^{\omega} \cdot$ to denote the existence of a T-labelled infinite execution. $(W, \mathcal{S}) \xrightarrow{T}{}^{*} (\textbf{skip}, \_)$ represents a terminating execution that produces T. By using $\lfloor W \rfloor$, we append **end** at the end of each thread to explicitly mark the

Definition. An object Π satisfies P under a refinement mapping φ, $P_\varphi(\Pi)$, iff $\forall n, C_1, \ldots, C_n, \mathcal{S}, T.\ T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \| \ldots \| C_n), \mathcal{S}]\!] \wedge (\mathcal{S} \in dom(\varphi)) \implies P(T)$.

$$\mathcal{T}_\omega[\![W, \mathcal{S}]\!] \stackrel{\text{def}}{=} \{(\mathbf{spawn}, |W|) :: T \mid (\lfloor W \rfloor, \mathcal{S}) \stackrel{T}{\longmapsto}{}^{\omega} \cdot \vee (\lfloor W \rfloor, \mathcal{S}) \stackrel{T}{\longmapsto}{}^{*} (\mathbf{skip}, \_) \vee (\lfloor W \rfloor, \mathcal{S}) \stackrel{T}{\longmapsto}{}^{*} \mathbf{abort}\}$$

$$\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \| \ldots \| C_n \rfloor \stackrel{\text{def}}{=} \mathbf{let}\ \Pi\ \mathbf{in}\ (C_1; \mathbf{end}) \| \ldots \| (C_n; \mathbf{end})$$

$$|\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \| \ldots \| C_n| \stackrel{\text{def}}{=} n \qquad \mathsf{tnum}((\mathbf{spawn}, n) :: T) \stackrel{\text{def}}{=} n$$

$$\mathsf{pend\_inv}(T) \stackrel{\text{def}}{=} \{e \mid \exists i.\ e = T(i) \wedge \mathsf{is\_inv}(e) \wedge \neg \exists j.\ (j > i \wedge \mathsf{match}(e, T(j)))\}$$

$$\mathsf{prog\text{-}t}(T) \ \text{ iff } \ \forall i, e.\ e \in \mathsf{pend\_inv}(T(1..i)) \implies \exists j.\ j > i \wedge \mathsf{match}(e, T(j))$$

$$\mathsf{prog\text{-}s}(T) \ \text{ iff } \ \forall i, e.\ e \in \mathsf{pend\_inv}(T(1..i)) \implies \exists j.\ j > i \wedge \mathsf{is\_ret}(T(j))$$

$$\mathsf{abt}(T) \ \text{ iff } \ \exists i.\ \mathsf{is\_abt}(T(i))$$

$$\mathsf{sched}(T) \ \text{ iff } \ |T| = \omega \wedge \mathsf{pend\_inv}(T) \neq \emptyset \implies \exists e.\ e \in \mathsf{pend\_inv}(T) \wedge |(T|_{\mathsf{tid}(e)})| = \omega$$

$$\mathsf{fair}(T) \ \text{ iff } \ |T| = \omega \implies \forall \mathsf{t} \in [1..\mathsf{tnum}(T)].\ |(T|_{\mathsf{t}})| = \omega \vee \mathsf{last}(T|_{\mathsf{t}}) = (\mathsf{t}, \mathbf{term})$$

$$\mathsf{iso}(T) \ \text{ iff } \ |T| = \omega \implies \exists \mathsf{t}, i.\ (\forall j.\ j \geq i \implies \mathsf{tid}(T(j)) = \mathsf{t})$$

$$\mathsf{wait\text{-}free} \ \text{ iff } \ \mathsf{sched} \implies \mathsf{prog\text{-}t} \vee \mathsf{abt} \qquad \mathsf{starvation\text{-}free} \ \text{ iff } \ \mathsf{fair} \implies \mathsf{prog\text{-}t} \vee \mathsf{abt}$$

$$\mathsf{lock\text{-}free} \ \text{ iff } \ \mathsf{sched} \implies \mathsf{prog\text{-}s} \vee \mathsf{abt} \qquad \mathsf{deadlock\text{-}free} \ \text{ iff } \ \mathsf{fair} \implies \mathsf{prog\text{-}s} \vee \mathsf{abt}$$

$$\mathsf{obstruction\text{-}free} \ \text{ iff } \ \mathsf{sched} \wedge \mathsf{iso} \implies \mathsf{prog\text{-}t} \vee \mathsf{abt}$$

Fig. 6. Formalizing Progress Properties

$$\mathsf{lock\text{-}free} \iff \mathsf{wait\text{-}free} \vee \mathsf{prog\text{-}s} \qquad \mathsf{starvation\text{-}free} \iff \mathsf{wait\text{-}free} \vee \neg\mathsf{fair}$$

$$\mathsf{obstruction\text{-}free} \iff \mathsf{lock\text{-}free} \vee \neg\mathsf{iso} \qquad \mathsf{deadlock\text{-}free} \iff \mathsf{lock\text{-}free} \vee \neg\mathsf{fair}$$

Fig. 7. Relationships between Progress Properties

termination of the thread. We also insert the spawning event $(\mathbf{spawn}, n)$ at the beginning of T, where n is the number of threads in W. Then we can use $\mathsf{tnum}(T)$ to get the number n, which is needed to define fairness, as shown below.

Before formulating each progress property over event traces, we first define some auxiliary properties in Figure 6. $\mathsf{prog\text{-}t}(T)$ guarantees that every method call in T eventually finishes. $\mathsf{prog\text{-}s}(T)$ guarantees that *some* pending method call finishes. Different from prog-t, the return event $T(j)$ in prog-s does not have to be a matching return of the pending invocation e. $\mathsf{abt}(T)$ says that T ends with a fault event.

There are three useful conditions on scheduling. The basic requirement for a good schedule is sched. If T is infinite and there exist pending calls, then at least one pending thread should be scheduled infinitely often. In fact, there are two possible reasons causing a method call of thread t to pend. Either t is no longer scheduled, or it is always scheduled but the method call never finishes. sched rules out the bad schedule where no thread with an invoked method is active. For instance, the following infinite trace does *not* satisfy sched.

$$\mathsf{div\_tids}(T) \stackrel{\text{def}}{=} \{\mathsf{t} \mid (\,|(T|_{\mathsf{t}})| = \omega\,)\}$$
$$\mathcal{O}_{\omega}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}_{\omega}[\![W,\mathcal{S}]\!]\}$$
$$\mathcal{O}_{i\omega}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}_{\omega}[\![W,\mathcal{S}]\!] \wedge \mathsf{iso}(T)\}$$
$$\mathcal{O}_{f\omega}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}_{\omega}[\![W,\mathcal{S}]\!] \wedge \mathsf{fair}(T)\}$$
$$\mathcal{O}_{t\omega}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{(\mathsf{get\_obsv}(T), \mathsf{div\_tids}(T)) \mid T \in \mathcal{T}_{\omega}[\![W,\mathcal{S}]\!]\}$$
$$\mathcal{O}_{ft\omega}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{(\mathsf{get\_obsv}(T), \mathsf{div\_tids}(T)) \mid T \in \mathcal{T}_{\omega}[\![W,\mathcal{S}]\!] \wedge \mathsf{fair}(T)\}$$

Fig. 8. Generation of Complete Event Traces

$$(\mathsf{t}_1, f_1, n_1) :: (\mathsf{t}_2, f_2, n_2) :: (\mathsf{t}_1, \mathbf{obj}) :: (\mathsf{t}_3, \mathbf{clt}) :: (\mathsf{t}_3, \mathbf{clt}) :: (\mathsf{t}_3, \mathbf{clt}) :: \ldots$$

If T is infinite, $\mathsf{fair}(T)$ requires every non-terminating thread be scheduled infinitely often; and $\mathsf{iso}(T)$ requires eventually only one thread be scheduled. We can see that a fair schedule is a good schedule satisfying sched.

At the bottom of Figure 6 we define the progress properties formally. We omit the parameter T in the formulae to simplify the presentation. An event trace T is wait-free (*i.e.*, $\mathsf{wait\text{-}free}(T)$ holds) if under the good schedule sched, it guarantees prog-t unless it ends with a fault. $\mathsf{lock\text{-}free}(T)$ is similar except that it guarantees prog-s. Starvation-freedom and deadlock-freedom guarantee prog-t and prog-s under fair scheduling. Obstruction-freedom guarantees prog-t if some pending thread is always scheduled (sched) and runs in isolation (iso).

Figure 7 contains lemmas that relate progress properties. For instance, an event trace is starvation-free, iff it is wait-free or not fair. These lemmas give us the relationship lattice in Figure 1. To close the lattice, we also define a progress property in the sequential setting. *Sequential termination* guarantees that every method call must finish in a trace produced by a sequential client. The formal definition is given in the companion TR [14], and we prove that it is implied by each of the five progress properties for concurrent objects.

5 Equivalence to Contextual Refinements

We extend the basic contextual refinement in Definition 3 to observe progress as well as linearizability. For each progress property, we carefully choose the observable behaviors at the concrete and the abstract levels.

5.1 Observable Behaviors

In Figure 8, we define various observable behaviors for the termination-sensitive contextual refinements.

We use $\mathcal{O}_{\omega}[\![W,\mathcal{S}]\!]$ to represent the set of observable event traces produced by complete executions of $(W,\mathcal{S})$. Recall that $\mathsf{get\_obsv}(T)$ gets the sub-trace of T consisting of all the observable events only. Unlike the prefix-closed set $\mathcal{O}[\![W,\mathcal{S}]\!]$, this definition utilizes $\mathcal{T}_{\omega}[\![W,\mathcal{S}]\!]$ (see Figure 6) whose event traces are all complete and could be infinite. Thus it allows us to observe divergence of the

Table 2. Contextual Refinements $\Pi \sqsubseteq^{P}_{\varphi} \Pi_A$ for Progress Properties P

P	wait-free	lock-free	obstruction-free	deadlock-free	starvation-free
$\Pi \sqsubseteq^{P}_{\varphi} \Pi_A$	$\mathcal{O}_{t\omega} \subseteq \mathcal{O}_{t\omega}$	$\mathcal{O}_{\omega} \subseteq \mathcal{O}_{\omega}$	$\mathcal{O}_{i\omega} \subseteq \mathcal{O}_{\omega}$	$\mathcal{O}_{f\omega} \subseteq \mathcal{O}_{\omega}$	$\mathcal{O}_{ft\omega} \subseteq \mathcal{O}_{t\omega}$

whole program. $\mathcal{O}_{i\omega}$ and $\mathcal{O}_{f\omega}$ take the complete observable traces of *isolating* and fair executions respectively. Here $\mathsf{iso}(T)$ and $\mathsf{fair}(T)$ are defined in Figure 6.

We could also observe divergence of individual threads rather than the whole program. We define $\mathsf{div\_tids}(T)$ to collect the set of threads that diverge in the trace T. Then we write $\mathcal{O}_{t\omega}[\![W, \mathcal{S}]\!]$ to get both the observable behaviors and the diverging threads in the complete executions. $\mathcal{O}_{ft\omega}[\![W, \mathcal{S}]\!]$ is defined similarly but considers fair executions only.

More on divergence. In general, divergence means non-termination. For example, we could say that the following two-threaded program (5.1) must diverge since it never terminates.

```
x := x + 1;    ||    while(true) skip;          (5.1)
```

But for individual threads, divergence is not equivalent to non-termination, since a non-terminating thread may either have an infinite execution or simply be not scheduled from some point due to unfair scheduling. We view only the former case as divergence. For instance, in an unfair execution, the left thread of (5.1) may never be scheduled and hence it has no chance to terminate. It does not diverge. Similarly, for the following program (5.2),

```
while(true) skip;    ||    while(true) skip;     (5.2)
```

the whole program must diverge, but it is possible that a single thread does not diverge in an execution.

5.2 New Contextual Refinements and Equivalence Results

In Table 2, we summarize the definitions of the termination-sensitive contextual refinements. Each new contextual refinement follows the basic one in Definition 3 but takes different observable behaviors as specified in Table 2. For example, the contextual refinement for wait-freedom is formally defined as follows:

$$\Pi \sqsubseteq^{\mathsf{wait\text{-}free}}_{\varphi} \Pi_A \;\; \text{iff} \;\; (\forall n, C_1, \ldots, C_n, \mathcal{S}, \mathcal{S}_a.\ (\varphi(\mathcal{S}) = \mathcal{S}_a) \Longrightarrow \mathcal{O}_{t\omega}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), \mathcal{S})]\!] \subseteq \mathcal{O}_{t\omega}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), \mathcal{S}_a]\!]).$$

Theorem 5 says that linearizability with a progress property P together is equivalent to the corresponding contextual refinement $\sqsubseteq^{P}_{\varphi}$.

Theorem 5 (Equivalence). *$\Pi \preceq_{\varphi} \Pi_A \wedge P_{\varphi}(\Pi) \iff \Pi \sqsubseteq^{P}_{\varphi} \Pi_A$, where P is* wait-free, lock-free, obstruction-free, deadlock-free *or* starvation-free.

Here we assume the object specification Π_A is *total*, *i.e.*, the abstract operations never block. We provide the proofs of our equivalence results in the TR [14].

The contextual refinement for wait-freedom takes $\mathcal{O}_{t\omega}$ at both the concrete and the abstract levels. The divergence of individual threads as well as I/O events are treated as observable behaviors. The intuition of the equivalence is as

follows. Since a wait-free object Π guarantees that every method call finishes, we have to blame the client code itself for the divergence of a thread using Π. That is, even if the thread uses the abstract object Π_A, it must still diverge.

As an example, consider the client program (2.1). Intuitively, for any execution in which the client uses the abstract operations, only the right thread $\mathtt{t}_2$ diverges. Thus $\mathcal{O}_{t\omega}$ of the abstract program is a singleton set $\{(\epsilon, \{\mathtt{t}_2\})\}$. When the client uses the wait-free object in Figure 2(a), its $\mathcal{O}_{t\omega}$ set is still $\{(\epsilon, \{\mathtt{t}_2\})\}$. It does not produce more observable behaviors. But if it uses a non-wait-free object (such as the one in Figure 2(b)), the left thread $\mathtt{t}_1$ does not necessarily finish. The $\mathcal{O}_{t\omega}$ set becomes $\{(\epsilon, \{\mathtt{t}_2\}), (\epsilon, \{\mathtt{t}_1, \mathtt{t}_2\})\}$. It produces more observable behaviors than the abstract client, breaking the contextual refinement. Thanks to observing div_tids that collects the diverging threads, we can rule out non-wait-free objects which may cause more threads to diverge.

$\Pi \sqsubseteq_{\varphi}^{\text{lock-free}} \Pi_A$ takes coarser observable behaviors. We observe the divergence of the whole client program by using $\mathcal{O}_\omega$ at both the concrete and the abstract levels. Intuitively, a lock-free object Π ensures that some method call will finish, thus the client using Π diverges only if there are an infinite number of method calls. Then it must also diverge when using the abstract object Π_A.

For example, consider the client (2.1). The whole client program diverges in every execution both when it uses the lock-free object in Figure 2(b) and when it uses the abstract one. The $\mathcal{O}_\omega$ set of observable behaviors is $\{\epsilon\}$ at both levels. On the other hand, the following client must terminate and print out both 1 and 2 in every execution. The $\mathcal{O}_\omega$ set is $\{1::2::\epsilon, 2::1::\epsilon\}$ at both levels.

```
inc(); print(1);   ||   dec(); print(2);        (5.3)
```

Instead, if the client (5.3) uses the non-lock-free object in Figure 2(c), it may diverge and nothing is printed out. The $\mathcal{O}_\omega$ set becomes $\{\epsilon, 1::2::\epsilon, 2::1::\epsilon\}$, which contains more behaviors than the abstract side. Thus $\Pi \sqsubseteq_{\varphi}^{\text{lock-free}} \Pi_A$ fails.

Obstruction-freedom ensures progress for isolating executions in which eventually only one thread is running. Correspondingly, $\Pi \sqsubseteq_{\varphi}^{\text{obstruction-free}} \Pi_A$ restricts our considerations to isolating executions. It takes $\mathcal{O}_{i\omega}$ at the concrete level and $\mathcal{O}_\omega$ at the abstract level.

To understand the equivalence, consider the client (5.3) again. For isolating executions with the obstruction-free object in Figure 2(c), it *must* terminate and print out both 1 and 2. The $\mathcal{O}_{i\omega}$ set at the concrete level is $\{1::2::\epsilon, 2::1::\epsilon\}$, the same as the set $\mathcal{O}_\omega$ of the abstract side. Non-obstruction-free objects in general do not guarantee progress for some isolating executions. If the client uses the object in Figure 2(d) or (e), the $\mathcal{O}_{i\omega}$ set is $\{\epsilon, 1::2::\epsilon, 2::1::\epsilon\}$, not a subset of the abstract $\mathcal{O}_\omega$ set. The undesired empty observable trace is produced by unfair executions, where a thread acquires the lock and gets suspended and then the other thread would keep requesting the lock forever (it is executed in isolation).

$\Pi \sqsubseteq_{\varphi}^{\text{deadlock-free}} \Pi_A$ uses $\mathcal{O}_{f\omega}$ at the concrete side, ruling out undesired divergence caused by unfair scheduling. For the client (5.3) with the object in Figure 2(d) or (e), its $\mathcal{O}_{f\omega}$ set is same as the set $\mathcal{O}_\omega$ at the abstract level.

For $\Pi \sqsubseteq_{\varphi}^{\text{starvation-free}} \Pi_A$, we still consider only fair executions at the concrete level (similar to deadlock-freedom), but observe the divergence of individual

threads rather than the whole program (similar to wait-freedom). It uses $\mathcal{O}_{ft\omega}$ at the concrete side and $\mathcal{O}_{t\omega}$ at the abstract level. For the client (5.3) with the starvation-free object in Figure 2(e), no thread diverges in any fair execution. Then the set $\mathcal{O}_{ft\omega}$ of observable behaviors is $\{(1::2::\epsilon, \emptyset), (2::1::\epsilon, \emptyset)\}$, which is same as the set $\mathcal{O}_{t\omega}$ at the abstract level.

Observing threaded divergence allows us to distinguish starvation-free objects from deadlock-free objects. Consider the client (2.1). Under fair scheduling, we know only the right thread $\mathtt{t}_2$ would diverge when using the starvation-free object in Figure 2(e). The set $\mathcal{O}_{ft\omega}$ is $\{(\epsilon, \{\mathtt{t}_2\})\}$. It coincides with the abstract behaviors $\mathcal{O}_{t\omega}$. But when using the deadlock-free object of Figure 2(d), the $\mathcal{O}_{ft\omega}$ set becomes $\{(\epsilon, \{\mathtt{t}_2\}), (\epsilon, \{\mathtt{t}_1, \mathtt{t}_2\})\}$, breaking the contextual refinement.

6 Related Work and Conclusion

There is a large body of work discussing the five progress properties and the contextual refinements individually. Our work in contrast studies their relationships, which have not been considered much before.

Gotsman and Yang [6] propose a new linearizability definition that preserves lock-freedom, and suggest a connection between lock-freedom and a termination-sensitive contextual refinement. We do not redefine linearizability here. Instead, we propose a unified framework to systematically relate all the five progress properties plus linearizability to various contextual refinements.

Herlihy and Shavit [10] informally discuss all the five progress properties. Our definitions in Section 4 mostly follow their explanations, but they are more formal and close the gap between program semantics and their history-based interpretations. We also notice that their obstruction-freedom is inappropriate for some examples (see TR [14]), and propose a different definition that is closer to the common intuition [9]. In addition, we relate the progress properties to contextual refinements, which consider the extensional effects on client behaviors.

Fossati *et al.* [5] propose a uniform approach in the π-calculus to formulate both the standard progress properties and their observational approximations. Their technical setting is completely different from ours. Also, their observational approximations for lock-freedom and wait-freedom are strictly weaker than the standard notions. Their deadlock-freedom and starvation-freedom are not formulated, and there is no observational approximation given for obstruction-freedom. In comparison, our framework relates each of the five progress properties (plus linearizablity) to an *equivalent* contextual refinement.

There are also formulations of progress properties based on temporal logics. For example, Petrank *et al.* [15] formalize the three non-blocking properties and Dongol [3] formalize all the five progress properties, using linear temporal logics. Those formulations make it easier to do model checking (*e.g.*, Petrank *et al.* [15] also build a tool to model check a variant of lock-freedom), while our contextual refinement framework is potentially helpful for modular Hoare-style verification.

Conclusion. We have introduced a contextual refinement framework to unify various progress properties. For linearizable objects, each progress property is

equivalent to a specific termination-sensitive contextual refinement, as summarized in Table 1. The framework allows us to verify safety and liveness properties of client programs at a high abstraction level by replacing concrete method implementations with abstract operations. It also makes it possible to borrow ideas from existing proof methods for contextual refinements to verify linearizability and a progress property together, which we leave as future work.

Acknowledgments. We would like to thank anonymous referees for their helpful suggestions and comments. This work is supported in part by China Scholarship Council, NSFC grants 61073040 and 61229201, NCET grant NCET-2010-0984, and the Fundamental Research Funds for the Central Universities (Grant No. WK0110000018). It is also supported in part by DARPA grants FA8750-10-2-0254 and FA8750-12-2-0293, ONR grant N000141210478, and NSF grants 0915888 and 1065451. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

1. Aspnes, J., Herlihy, M.: Wait-free data structures in the asynchronous PRAM model. In: SPAA, pp. 340–349 (1990)
2. Birkedal, L., Sieczkowski, F., Thamsborg, J.: A concurrent logical relation. In: CSL, pp. 107–121 (2012)
3. Dongol, B.: Formalising progress properties of non-blocking programs. In: Liu, Z., Kleinberg, R.D. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 284–303. Springer, Heidelberg (2006)
4. Filipovic, I., O'Hearn, P., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. Theor. Comput. Sci. 411(51-52), 4379–4398 (2010)
5. Fossati, L., Honda, K., Yoshida, N.: Intensional and extensional characterisation of global progress in the π-calculus. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 287–301. Springer, Heidelberg (2012)
6. Gotsman, A., Yang, H.: Liveness-preserving atomicity abstraction. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 453–465. Springer, Heidelberg (2011)
7. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. 13(1), 124–149 (1991)
8. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: ICDCS, pp. 522–529 (2003)
9. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (April 2008)
10. Herlihy, M., Shavit, N.: On the nature of progress. In: Fernàndez Anta, A., Lipari, G., Roy, M. (eds.) OPODIS 2011. LNCS, vol. 7109, pp. 313–328. Springer, Heidelberg (2011)
11. Herlihy, M., Wing, J.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
12. Liang, H., Feng, X.: Modular verification of linearizability with non-fixed linearization points. In: PLDI (to appear, 2013)
13. Liang, H., Feng, X., Fu, M.: A rely-guarantee-based simulation for verifying concurrent program transformations. In: POPL, pp. 455–468 (2012)
14. Liang, H., Hoffmann, J., Feng, X., Shao, Z.: The extended version of the present paper (2013), `http://kyhcs.ustcsz.edu.cn/relconcur/prog`
15. Petrank, E., Musuvathi, M., Steensgaard, B.: Progress guarantee for parallel programs via bounded lock-freedom. In: PLDI, pp. 144–154 (2009)