Progress of Concurrent Objects with Partial Methods

HONGJIN LIANG and XINYU FENG[*], Nanjing University, China and University of Science and Technology of China, China

Various progress properties have been proposed for concurrent objects, such as wait-freedom, lock-freedom, starvation-freedom and deadlock-freedom. However, none of them applies to concurrent objects with partial methods, i.e., methods that are supposed *not* to return under certain circumstances. A typical example is the `lock_acquire` method, which must *not* return when the lock has already been acquired.

In this paper we propose two new progress properties, partial starvation-freedom (PSF) and partial deadlock-freedom (PDF), for concurrent objects with partial methods. We also design four patterns to write abstract specifications for PSF or PDF objects under strongly or weakly fair scheduling, so that these objects contextually refine the abstract specifications. Our Abstraction Theorem shows the equivalence between PSF (or PDF) and the progress-aware contextual refinement. Finally, we generalize the program logic LiLi to have a new logic to verify the PSF (or PDF) property and linearizability of concurrent objects.

CCS Concepts: • **Theory of computation → Hoare logic**; **Program specifications**; **Program verification**; **Abstraction**;

Additional Key Words and Phrases: Concurrent Objects, Progress, Refinement, Partial Methods

ACM Reference Format:
Hongjin Liang and Xinyu Feng. 2018. Progress of Concurrent Objects with Partial Methods. *Proc. ACM Program. Lang.* 2, POPL, Article 20 (January 2018), 31 pages. https://doi.org/10.1145/3158108

1 INTRODUCTION

A concurrent object consists of shared data and a set of methods which provide an interface for client threads to access the shared data. Linearizability [Herlihy and Wing 1990] has been used as a standard definition of the correctness of concurrent object implementations. It describes safety and functionality, but has no requirement about termination of object methods. Various progress properties, such as wait-freedom, lock-freedom, starvation-freedom and deadlock-freedom, have been proposed to specify termination of object methods. In their textbook Herlihy and Shavit [2008] give a systematic introduction of these properties.

Recent work [Filipović et al. 2009] has shown the equivalence between linearizability and a contextual refinement. The result has been further extended [Gotsman and Yang 2011; Liang and Feng 2016; Liang et al. 2013] to show that, when progress properties are taken into account, one may have the corresponding progress-aware contextual refinement to reestablish the equivalence. The equivalence results allow us to build abstractions for linearizable objects so that safety and progress of the client code can be reasoned about at a more abstract level.

[*]Corresponding author.

Authors' address: Hongjin Liang, hongjin@nju.edu.cn; Xinyu Feng, xyfeng@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, 163 Xianlin Road, Nanjing, 210023, China , University of Science and Technology of China, 443 Huangshan Road, Hefei, 230000, China.

2475-1421/2018/1-ART20
https://doi.org/10.1145/3158108

However, *none* of these progress-related results applies to concurrent objects with partial methods! A method is *partial* if it is supposed *not* to return under certain circumstances. A typical example is the `lock_acquire` method, which must *not* return when the lock has already been acquired. Concurrent objects with partial methods simply do not satisfy any of the aforementioned progress properties, and people do not know how to give progress-aware abstract specifications for them either. The existing studies on progress properties and progress-aware contextual refinements have been limited to concurrent objects with total specifications.

As an awkward consequence, we cannot treat lock implementations as objects when we study progress of concurrent objects. Instead, we have to treat `lock_acquire` and `lock_release` as *internal* functions of other lock-based objects. Also, without a proper progress-aware abstraction for locks, we have to redo the verification of `lock_acquire` and `lock_release` when they are used in different contexts [Liang and Feng 2016], which makes the verification process complex and painful. Note that locks are not the only objects with partial methods. Concurrent sets, stacks and queues may also have methods that intend to block. For instance, it may be sensible for a thread attempting to pop from an empty stack to block, waiting until another thread pushes an item. The reasoning about these objects suffers from the same problems too when progress is concerned.

We face the following key challenges to address these problems.

- We need definitions of new progress properties for these objects, and the definitions need to describe the situations in which permanent blocking is allowed. It is important to note that, although deadlock-freedom and starvation-freedom have been used as progress properties for "blocking" algorithms [Herlihy and Shavit 2008], they allow permanent blocking only when the scheduling is unfair. They can specify concurrent objects implemented using locks, but they do not apply to lock objects themselves. For objects like locks, blocking may also be caused by inappropriate method invocations by the client. For instance, if a thread of the client fails to call `lock_release` after acquiring the lock, the calls to `lock_acquire` by other threads will be always blocked. Similarly, for a stack object with a partial `pop` method, if no client threads call `push`, the calls to `pop` will be permanently blocked at an empty stack. The question is, how to distinguish the blocking behaviors caused by "bad" clients with those caused by bad object implementations, and blame the objects only for the blocking in the latter case.
- The abstractions for objects with partial methods should be able to distinguish the objects with different progress guarantees under different scheduling conditions. A natural abstraction for partial methods is the blocking primitive **await**$(B)\{C\}$. It is blocked if B does not hold, and executes C atomically if B holds (in this case, we say the code **await**$(B)\{C\}$ is *enabled*). A specification in the form of **await**$(B)\{C\}$ can characterize both the atomicity of the functionality and the fact that the method is partial. However, it is not sufficient to serve as a progress-aware abstraction for the following two reasons.
  - Different implementations of the same **await** block may exhibit different progress properties, requiring different abstractions. For instance, the ticket lock algorithm [Mellor-Crummey and Scott 1991] has stronger progress guarantees than the test-and-set lock algorithm [Herlihy and Shavit 2008]. Therefore when progress is concerned it is impossible to use the same partial specification (e.g., **await**`(l=0){l := cid}`, where `cid` is the ID of the current thread) as an abstraction for the `lock_acquire` methods in both algorithms (even though it may work for both if we consider linearizability only).
  - Even the same implementation may require different abstractions for different scheduling. The blocking primitive **await**$(B)\{C\}$ behaves differently under strongly fair and weakly fair scheduling. The former ensures the execution of the primitive as long as it is enabled a

sufficient number of times, but the latter requires the primitive to be *always* enabled to ensure its execution. On the other hand, the distinction between strong and weak fairness is meaningful only if there are blocking primitives. A low-level program consisting of non-blocking primitive instructions only (like most machine instructions) behaves the same under both scheduling. Such a program cannot have the same abstraction with blocking primitives under different scheduling.

As a result, if we consider m kinds of progress properties (e.g., to distinguish ticket locks and test-and-set locks) and the 2 choices of strongly fair and weakly fair scheduling, we may need as many as $2 \times m$ kinds of abstractions for the same functionality. Can we find general patterns for these abstractions?

In this paper, we specify and verify progress of concurrent objects with partial methods. We define progress properties and abstraction patterns under strongly and weakly fair scheduling. Then we prove that given a linearizable object implementation Π, the contextual refinement between Π and its abstraction Π' under a certain kind of fair scheduling is equivalent to the progress property of Π. We also provide a program logic to verify the contextual refinement between Π and Π', which ensures linearizability and the progress property of Π.

Our work is based on earlier work on abstraction for concurrent objects and concurrency verification, but makes the following new contributions:

- We propose progress properties, *partial starvation-freedom (PSF)* and *partial deadlock-freedom (PDF)*, for concurrent objects with partial methods. They identify the execution scenarios in which the partial methods are blocked due to inappropriate invocation sequences made by "bad" clients, and allow the object methods to be blocked permanently in these special scenarios. Ticket locks and test-and-set locks satisfy PSF and PDF respectively. Traditional starvation-freedom and deadlock-freedom for objects with total methods can be viewed as specializations of PSF and PDF respectively, if we view total methods as special cases of partial ones that are always enabled to return.
- We design four general patterns for abstractions for concurrent objects with PSF and PDF progress properties under strongly and weakly fair scheduling, respectively. We start with the basic blocking primitive **await**$(B)\{C\}$ and define syntactic wrappers that transform it into non-atomic object specifications which can be refined by the object implementations in the progress-aware contextual refinement. We give distinguished wrappers for different combinations of fairness and progress properties.
- We prove the equivalence between PSF (or PDF) and the progress-aware contextual refinement, where the abstraction is generated by the wrapper, under strong or weak fairness. The equivalence results (called the abstraction theorem) allow us to verify safety and liveness properties of client programs at a high abstraction level, by replacing concrete object implementations with the specifications. Using the natural transitivity of the contextual refinement, it is also possible to verify linearizability and PSF (or PDF) of nested concurrent objects.
- We design a program logic to verify objects with PSF or PDF progress properties. Our logic is a generalization of the LiLi logic for starvation-free and deadlock-free objects [Liang and Feng 2016]. It also provides inference rules for the **await**$(B)\{C\}$ statement under strong and weak fairness, so that **await** commands can also be used in object implementations. The soundness of our logic ensures the progress-aware contextual refinement, and linearizability and PSF (or PDF) under different fairness. We have applied the program logic to verify ticket locks [Mellor-Crummey and Scott 1991], test-and-set locks [Herlihy and Shavit 2008], bounded partial queues with two locks [Herlihy and Shavit 2008] and Treiber stacks [Treiber 1986] with possibly blocking pop methods.

```
L_initialize(){  l := 0; }

L_acq(){
  local b := false;
  while(!b){ b := cas(&l, 0, cid); }
}

L_rel(){
  l := 0;
}
```

(a) test-and-set lock implementation

```
inc(){ L_acq(); x:=x+1; L_rel(); }
```

(b) counter with a test-and-set lock

```
tkL_initialize(){ owner := 0; next := 0; }

tkL_acq(){
  local i, o;
  i := getAndInc(&next);
  o := owner; while(i!=o){ o := owner; }
}

tkL_rel(){
  owner := owner + 1;
}
```

(c) ticket lock implementation

```
inc_tkL(){ tkL_acq(); x:=x+1; tkL_rel(); }
```

(d) counter with a ticket lock

```
INC(){x:=x+1;}
```

(e) atomic spec. INC

Fig. 1. Counters with locks.

In the rest of this paper, we first give an informal overview of the background and our key ideas in Sec. 2. Then we introduce the object language in Sec. 3, and linearizability and the basic contextual refinement in Sec. 4. We propose our new progress properties in Sec. 5, and give the progress-aware contextual refinement and the abstraction theorem in Sec. 6. We present the logic in Sec. 7 and show the examples we have verified in Sec. 8. Finally, we discuss related work and conclude in Sec. 9.

2 BACKGROUND AND OVERVIEW OF KEY IDEAS

Below we first give an overview of linearizability, starvation-freedom, deadlock-freedom and contextual refinement. Then we analyze the problems in defining progress of concurrent objects with partial methods, and explain our solutions informally.

2.1 Background

A concurrent object usually satisfies linearizability, a standard safety criterion, and certain progress property, describing when and how method calls of the object are guaranteed to terminate.

Linearizability. A concurrent object is linearizable, if each method call appears to take effect instantaneously at some moment between its invocation and return [Herlihy and Wing 1990]. Intuitively, linearizability requires the implementation of each method to have the same effect as an atomic specification.

Consider the two implementations of the counter object in Fig. 1(b) and (d). We assume that every primitive command is executed atomically. A counter provides a method `inc` for incrementing the shared data `x`. Both implementations use locks to synchronize the increments. Intuitively they have the same effect as the atomic specification `INC()` in Fig. 1(e), so they are linearizable.

The locks themselves could also be viewed as standalone objects. For instance, the test-and-set lock object in Fig. 1(a) provides the methods `L_acq` and `L_rel` for a thread to acquire and release the lock `l`. Here `cid` represents the current thread's ID, which is a positive number. The counter's implementation code in Fig. 1(b) can be viewed as a client of this lock object. The lock object is linearizable, because `L_acq` and `L_rel` both update `l` atomically (if they indeed return). They produce the same effects as the atomic operations `L_ACQ` and `L_REL` (defined below), respectively:

```
L_ACQ(){ l := cid; }      L_REL(){ l := 0; }                (2.1)
```

However, linearizability does not characterize progress properties of the object implementations. For instance, the following counter object is still linearizable, even if its method never terminates.

```
inc'(){ L_acq(); x:=x+1; L_rel(); while(true) skip; }
```

Progress properties. Various progress properties have been proposed for concurrent objects, such as wait-freedom and lock-freedom for non-blocking implementations, and starvation-freedom and deadlock-freedom for lock-based implementations. These properties describe conditions under which a method call is guaranteed to successfully finish in an execution. The two implementations of the counter in Fig. 1(b) and (d) satisfy deadlock-freedom and starvation-freedom respectively.

We use the definitions given by Herlihy and Shavit [2011]. Both deadlock-freedom and starvation-freedom assume fair scheduling, i.e., every thread gets eventually executed. For the counters in Fig. 1(b) and (d), fairness ensures that every thread holding the lock will eventually release the lock.

Deadlock-freedom requires "minimal progress" in fair executions, i.e., there always exists some method call which can finish under fair scheduling, while starvation-freedom requires "maximal progress" in fair executions, i.e., every method call should eventually finish under fair scheduling.

The counter in Fig. 1(b) is deadlock-free, because the test-and-set lock (see Fig. 1(a)) guarantees that eventually some thread will succeed in getting the lock via the **cas** instruction at line 2, and hence the method call of `inc` in that thread will eventually finish. It is not starvation-free, because there might be a thread that continuously fails to acquire the lock. For the following client program (2.2), the **cas** instruction executed by the left thread could always fail if the right thread infinitely often acquires the lock.

```
inc(); print(1);   ||   while(true) inc();                  (2.2)
```

The counter in Fig. 1(d) implemented with a ticket lock is starvation-free. Figure 1(c) shows the details of the ticket lock implementation. It uses two shared variables `owner` and `next`, which are equal initially. The threads attempting to acquire the lock form a waiting queue. In `tkL_acq`, a thread first atomically increments `next` and reads its old value to a local variable `i` (line 2). It waits until the lock's `owner` equals its ticket number `i` (line 3), then it acquires the lock. In `tkL_rel`, the thread releases the lock by incrementing `owner` (line 4). Then the next waiting thread (the thread with ticket number `i+1`, if there is one) can acquire the lock. We can see that the ticket lock implementation ensures the first-come-first-served property, and hence every thread calling `inc_tkL` can eventually acquire the lock and finish its method call.

Deadlock-freedom and starvation-freedom are progress properties for the so-called "blocking implementations" [Herlihy and Shavit 2008], such as the counters in Fig. 1(b) and (d), where a thread holding a lock will block other threads requesting the lock. However, they do not apply to lock objects, e.g., the ones in Fig. 1(a) and (c). We will explain the problems in detail in Sec. 2.2.

Contextual refinement and the abstraction theorem. It is difficult to use linearizability and progress properties directly in modular verification of client programs of an object, because their definitions fail to describe how the client behaviors are affected. To verify clients, we would like to abstract away the details of the object implementation. This requires a notion of object correctness, telling us that the client behaviors will not change when we replace the object methods' implementations with the corresponding abstract operations (as specifications).

Contextual refinement gives the desired notion of correctness. Informally, an object implementation Π is a contextual refinement of a (more abstract) implementation Π', if every observable behavior of any client program using Π can also be observed when the client uses Π' instead. Then, when verifying a client of Π, we can soundly replace Π with its abstraction Π'.

There has been much work (e.g., [Filipović et al. 2009; Gotsman and Yang 2012; Liang et al. 2013]) studying abstraction theorems, which relate linearizability and progress properties with contextual refinements. It has been proved that linearizability of Π is equivalent to a contextual refinement between Π and its *atomic specification* Γ, where the observable behaviors are finite traces of I/O events. When taking progress properties into account, the corresponding contextual refinement should be sensitive to termination or divergence (non-termination). For instance, deadlock-freedom or starvation-freedom of linearizable objects is shown equivalent to a contextual refinement which observes (possibly infinite) full traces of I/O events in fair executions. Then, a client which diverges with Π in a fair execution must also have a diverging execution when using the abstraction Π'. Deadlock-free and starvation-free objects could be distinguished by different abstractions. The abstraction for starvation-free objects is the atomic specification Γ, while for deadlock-free ones the abstraction has to be non-atomic [Liang and Feng 2016].

The counter implementation `inc_tkL()` in Fig. 1(d) is a progress-aware contextual refinement of the atomic counter `INC` in Fig. 1(e), but `inc()` in Fig. 1(b) is not. To see the difference, consider the client program (2.2). Under fair scheduling, the client calling `inc()` may generate an empty I/O event trace because it may not print out 1. However, the empty trace cannot be generated when replacing `inc()` with `inc_tkL()` or `INC()`, because the resulting program must print out 1.

2.2 Problems and Our Solutions

The existing progress properties and the corresponding contextual refinement are proposed for concurrent objects with total methods only, i.e., methods that should always return when executed sequentially. They do not apply to objects with partial methods, such as the lock objects in Fig. 1(a) and (c), which intend to block at certain situations. We have outlined the key challenges in reasoning about progress properties of objects with partial methods in Sec. 1. We give more detailed explanations here.

2.2.1 *Atomic Specifications Need to Be Partial.*

The specifications defined in (2.1) can characterize the atomic behaviors of lock objects, but they fail to specify that `L_ACQ` should be partial in the sense that it should be blocked when the lock is unavailable.

To address the problem, we introduce the *atomic partial specification* Γ, where each method specification is in the form of **await**$(B)\{C\}$. For the lock objects, we can define the atomic partial specification Γ as follows.

```
L_ACQ'(){  await (l = 0) { l := cid };  }        L_REL(){ l := 0; }     (2.3)
```

The **await** block naturally specifies the atomicity of method functionality, just like the traditional atomic specification $\langle C\rangle$ (which can be viewed as syntactic sugar for **await**(true)$\{C\}$), therefore Γ may serve as a specification for linearizable objects. It also shows the fact that the object method is partial, with explicit specification of the enabling condition B. Below we use the atomic partial specification as the starting point to characterize the progress of objects.

2.2.2 *Deadlock-Freedom and Starvation-Freedom Do Not Apply.*

We need new progress properties for objects with partial methods. Consider the following client program (2.4) using the test-and-set lock in Fig. 1(a). One of the method calls never finishes.

```
L_acq();   ||  L_acq();                                    (2.4)
```

Table 1. Client (2.5) with different locks. “Yes” means it must print out 1, “No” otherwise.

	spec. (2.3)	ticket locks (Fig. 1(c))	test-and-set locks (Fig. 1(a))
Strong Fairness	Yes	Yes	No
Weak Fairness	No	Yes	No

It shows that the test-and-set lock object does not satisfy the traditional deadlock-freedom or starvation-freedom property we just presented. Neither does the ticket lock object in Fig. 1(c).

The problem is that `L_acq` intends to block when the lock is not available. The non-termination in the above example (2.4) is just the intention of a correct lock implementation; otherwise the lock cannot guarantee mutual exclusion.

Our solution. We define two new progress properties for objects with partial methods, which we call partial starvation-freedom (PSF) and partial deadlock-freedom (PDF). PSF requires that in each fair execution trace by any client with the object Π, either each method invocation eventually returns (as required in starvation-freedom), or each pending method invocation must be continuously *disabled*. The latter case intuitively says that this non-termination is caused by the “bad” client, e.g., by inappropriate invocations of the methods. Similarly, PDF requires that in each fair execution trace by any client with the object Π, either there always *exists* a method invocation that eventually returns (as in deadlock-freedom), or each pending method invocation must be continuously disabled.

But how do we formally say that a method is disabled? When we informally say this, we actually refer to the enabling condition B in $\textbf{await}(B)\{C\}$ in the object's atomic partial specification Γ. However, we may not be able to infer such a condition from the concrete implementation Π. To address this problem, our definitions of $\mathsf{PSF}_\Gamma(\Pi)$ and $\mathsf{PDF}_\Gamma(\Pi)$ are parameterized with the specification Γ (the actual definitions take more parameters, as shown in Sec. 5).

We can prove the lock objects in Fig. 1(a) and (c) satisfy PDF and PSF respectively. We can also show that starvation-freedom and deadlock-freedom are special cases of our PSF and PDF respectively, by instantiating the parameter Γ with specifications in the form of $\textbf{await}(\mathrm{true})\{C\}$.

2.2.3 Atomic Partial Specifications Are Insufficient for Progress-Aware Abstractions. Although the atomic partial specification Γ describes the atomic functionality and the enabling condition of each method, it is insufficient to serve as a progress-aware abstraction for the following reasons.

First, the progress of the **await** command itself is affected by the fairness of scheduling, such as strong fairness and weak fairness.

- *Strong fairness*: Every thread which is infinitely often enabled will execute infinitely often. Then, $\textbf{await}(B)\{C\}$ is not executed only if B is continuously false after some point in the execution trace.
- *Weak fairness*: Every thread which is eventually always enabled will execute infinitely often. Then, $\textbf{await}(B)\{C\}$ may not be executed when B is infinitely often false. This fairness notion is weaker than strong fairness.

As a result, the choice of fair scheduling will affect the behaviors of a program or a specification with **await** commands. To see this, we consider the following client program (2.5).

$$[\_]_{\mathrm{ACQ}};\ [\_]_{\mathrm{REL}};\ \mathtt{print(1)};\quad \|\quad \mathtt{while(true)\{}\ [\_]_{\mathrm{ACQ}};\ [\_]_{\mathrm{REL}};\ \mathtt{\}} \qquad (2.5)$$

where $[\_]_{\mathrm{ACQ}}$ and $[\_]_{\mathrm{REL}}$ represent holes to be filled with method calls of lock acquire and release, respectively. Table 1 shows the behaviors of the client with different locks. If the client calls the abstract specifications in (2.3), it must execute `print(1)` under strongly fair scheduling, but may not do so under weakly fair scheduling. This is because the call of `L_ACQ'` could be infinitely

Table 2. Wrappers for atomic specifications.

	PSF	PDF
Strong Fairness	$\mathsf{wr}^{\mathsf{sfair}}_{\mathsf{PSF}}(\mathbf{await}(B)\{C\})$	$\mathsf{wr}^{\mathsf{sfair}}_{\mathsf{PDF}}(\mathbf{await}(B)\{C\})$
Weak Fairness	$\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PSF}}(\mathbf{await}(B)\{C\})$	$\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PDF}}(\mathbf{await}(B)\{C\})$

often enabled and infinitely often disabled in an execution, making its termination sensitive to the fairness of scheduling.

Also note that the two fairness notions coincide when the program does not contain blocking operations. Therefore, regardless of strongly or weakly fair scheduling, the client (2.5) using ticket locks always executes `print(1)`, but it may not do so if using test-and-set locks instead (see Table 1).

As a result, for the same object implementation, we may need *different abstractions under different scheduling*. As shown in Table 1, the specification (2.3) cannot serve as the specification of the test-and-set locks under both strong fairness and weak fairness.

Second, even under the same scheduling, *different implementations demonstrate different progress, therefore need different abstractions*. As shown in Table 1, the different lock implementations have different behaviors, even under the same scheduling.

For the above two reasons, we need different abstractions for different combinations of fairness and progress. For PSF and PDF under strong and weak fairness respectively, we may need four different abstractions. Can we systematically generate all of them?

Our solution. We define code wrappers over the basic blocking primitive $\mathbf{await}(B)\{C\}$ to generate the abstractions. The code wrappers are syntactic transformations that transform $\mathbf{await}(B)\{C\}$ into possibly non-atomic object specifications which can be refined by the object implementations in the progress-aware contextual refinement. As shown in Table 2, the four wrappers correspond to all combinations of fairness and progress. The definitions are shown in Sec. 6. Here we only give some high-level intuitions using the lock objects as examples.

First, we observe that the lock specification (2.3) can already serve as an abstraction for ticket locks under strong fairness, or for test-and-set locks under weak fairness. In general, the wrapper $\mathsf{wr}^{\mathsf{sfair}}_{\mathsf{PSF}}$ can be an identity function, i.e., the atomic partial specifications are already proper abstractions for PSF objects (not only for locks) under strong fairness. But $\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PDF}}$ is subtle. The atomic partial specifications are insufficient as abstractions for general PDF objects under weak fairness, which we will explain in detail in Sec. 6.

Second, as we have seen from Table 1, the lock specification (2.3) does not work for PSF locks under weak fairness nor for PDF locks under strong fairness. Then the role of the wrapper $\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PSF}}$ (or $\mathsf{wr}^{\mathsf{sfair}}_{\mathsf{PDF}}$) is to generate the same PSF (or PDF) behaviors even though the fairness of scheduling is weaker (or stronger).

To guarantee PSF, the idea is to create some kind of "fairness" on termination, i.e., every method call can get the chance to terminate. Given weakly fair scheduling, this requires the enabling condition of the abstraction to continuously remain true. As a result, a possible way to define $\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PSF}}$(`L_ACQ'`) is to keep a queue of threads requesting the lock, and a thread can acquire the lock only when it is at the head of the queue.

To support PDF under strongly fair scheduling, we have to allow non-termination even if the enabling condition is infinitely often true. For the client (2.5), the call of `L_ACQ'` in the specification (2.3) under strongly fair scheduling always terminates. Then $\mathsf{wr}^{\mathsf{sfair}}_{\mathsf{PDF}}$ needs to incorporate with some kind of delaying mechanisms, so that the termination of `L_ACQ'` of the left thread could be delayed every time when the right thread succeeds in acquiring the lock.

$$
\begin{array}{rlll}
(\mathit{MName}) & f, g \ldots & & \qquad (\mathit{PVar})\ \ x, y, z \ldots \\
(\mathit{Expr}) & E & ::= & x \mid n \mid E + E \mid \ldots \qquad (\mathit{BExp})\ \ B\ ::=\ \mathsf{true} \mid \mathsf{false} \mid E = E \mid \neg B \mid \ldots \\
(\mathit{Stmt}) & C & ::= & x := E \mid x := [E] \mid [E] := E \mid \mathbf{print}(E) \mid x := \mathbf{cons}(E, \ldots, E) \mid \mathbf{dispose}(E) \\
 & & \mid & \mathbf{skip} \mid x := f(E) \mid \mathbf{return}\ E \mid C; C \mid \mathbf{if}\ (B)\ C\ \mathbf{else}\ C \mid \mathbf{while}\ (B)\{C\} \\
 & & \mid & \mathbf{await}(B)\{C\} \\
(\mathit{ODecl}) & \Pi, \Gamma & ::= & \{f_1 \leadsto (\mathcal{P}_1, x_1, C_1), \ldots, f_n \leadsto (\mathcal{P}_n, x_n, C_n)\} \\
(\mathit{Prog}) & W & ::= & \mathbf{let}\ \Pi\ \mathbf{in}\ \hat{C}_1 \parallel \ldots \parallel \hat{C}_n \qquad (\mathit{Thrd})\ \ \hat{C}\ ::=\ C \mid \mathbf{end}
\end{array}
$$

Fig. 2. Syntax of the programming language.

2.2.4 Other Results. We also have the following new results in addition to the new progress properties and code wrappers.

Abstraction theorem. We prove the abstraction theorem, saying that our new progress properties PSF and PDF (together with linearizability) are equivalent to contextual refinements where the abstractions are generated by the corresponding wrappers. On the one hand, the theorem justifies the abstractions generated by our wrappers, showing that they are refined by linearizable and PSF (or PDF) object implementations. On the other hand, it also justifies our formulation of PSF and PDF by showing that they imply progress-aware contextual refinements.

The abstraction theorem also allows us to verify safety and progress properties of whole programs (consisting of clients and objects) in a modular way — after proving linearizability and PSF (or PDF) of an object Π with respect to its atomic partial specification Γ, we can replace Π with the abstraction generated by applying the corresponding wrapper over Γ, and then reason about properties of the whole program at the high abstraction level.

Program logic. Finally we design a program logic as the proof method for verifying PSF and PDF objects. It ensures linearizability and PSF (or PDF) of an object Π with respect to its atomic partial specification Γ. The logic is a generalization of our previous program logic LiLi for starvation-free and deadlock-free objects [Liang and Feng 2016], plus new inference rules for the **await** statement under strong and weak fairness. We will explain the details in Sec. 7.

3 THE LANGUAGE

Figure 2 shows the syntax of the language. A *program* W consists of an *object declaration* Π and n parallel threads $\hat{C}$ as *clients* sharing the object. To simplify the language, we assume there is only one object in each program. Each Π maps method names f_i to annotated method implementations $(\mathcal{P}_i, x_i, C_i)$, where x_i and C_i are the formal parameter and the method body respectively, and the assertion $\mathcal{P}_i$ is an annotated precondition over the object state to ensure the safe execution of the method. It is defined in Fig. 3 and is used in the operational semantics explained below. A thread $\hat{C}$ is either a command C, or an **end** flag marking termination of the thread. The commands include the standard ones used in separation logic, where $x := [E]$ and $[E] := E'$ read and write the heap at the location E respectively, and $x := \mathbf{cons}(E, \ldots, E)$ and $\mathbf{dispose}(E)$ allocate and free memory cells respectively. In addition, we have method call $(x := f(E))$ and return $(\mathbf{return}\ E)$ commands. The $\mathbf{print}(E)$ command generates *externally observable events*, which are used to define trace refinements in Sec. 4. The $\mathbf{await}(B)\{C\}$ command is the only *blocking* primitive in the language. It blocks the current thread if B does not hold, otherwise C is executed *atomically* together with the testing of B.

$$
\begin{array}{rccl@{\qquad}rccl}
(\mathit{ThrdID}) & \mathrm{t} & \in & \mathit{Nat} & (\mathit{Store}) & s, \mathbb{s} & \in & \mathit{PVar} \rightharpoonup \mathit{Val} \\
(\mathit{Heap}) & h, \mathbb{h} & \in & \mathit{Nat} \rightharpoonup \mathit{Val} & (\mathit{Data}) & \sigma, \Sigma & ::= & (s, h) \\
(\mathit{CallStk}) & \kappa, \mathbb{k} & ::= & \circ \mid (s_l, x, C) & (\mathit{ThrdPool}) & \mathcal{K}, \mathbb{K} & ::= & \{\mathrm{t}_1 \rightsquigarrow \kappa_1, \ldots, \mathrm{t}_n \rightsquigarrow \kappa_n\} \\
(\mathit{PState}) & \mathcal{S}, \mathbb{S} & ::= & (\sigma_c, \sigma_o, \mathcal{K}) & (\mathit{LState}) & \varsigma, \delta & ::= & (\sigma_c, \sigma_o, \kappa)
\end{array}
$$

$$
\begin{array}{rccl@{\qquad}rccl}
(\mathit{ExecCtxt}) & \mathbf{E} & ::= & [\,] \mid \mathbf{E}; C & & & & \\
(\mathit{Pre}) & \mathcal{P} & \in & \mathscr{P}(\mathit{Data}) & (\mathit{AbsFun}) & \varphi & \in & \mathit{Data} \rightharpoonup \mathit{Data}
\end{array}
$$

$$
\begin{array}{rccl}
(\mathit{Event}) & e & ::= & (\mathrm{t}, f, n) \mid (\mathrm{t}, \mathbf{ret}, n) \mid (\mathrm{t}, \mathbf{obj}) \mid (\mathrm{t}, \mathbf{obj}, \mathbf{abort}) \mid (\mathrm{t}, \mathbf{out}, n) \\
 & & \mid & (\mathrm{t}, \mathbf{clt}) \mid (\mathrm{t}, \mathbf{clt}, \mathbf{abort}) \mid (\mathrm{t}, \mathbf{term}) \mid (\mathbf{spawn}, n) \\
(\mathit{BIdSet}) & \Delta & \in & \mathscr{P}(\mathit{ThrdID}) \qquad\qquad (\mathit{PEvent}) \;\; \iota \;\; ::= \;\; (e, \Delta_c, \Delta_o) \\
(\mathit{ETrace}) & \mathcal{E} & ::= & \epsilon \mid e :: \mathcal{E} \quad \text{(co-inductive)} \qquad (\mathit{PTrace}) \;\; T \;\; ::= \;\; \epsilon \mid \iota :: T \quad \text{(co-inductive)}
\end{array}
$$

$$
\mathsf{en}(\hat{C}) \stackrel{\text{def}}{=} \begin{cases} B & \text{if } \exists \mathbf{E}, C'.\ \hat{C} = \mathbf{E}[\mathbf{await}(B)\{C'\}] \\ \mathsf{true} & \text{otherwise} \end{cases}
$$

$$
(\sigma_o, \kappa) \models B \;\text{ iff }\; [\![B]\!]_{((\sigma_o.s) \uplus (\kappa.s_l))} = \mathsf{true} \wedge \kappa \neq \circ \qquad\qquad \sigma_c \models B \;\text{ iff }\; [\![B]\!]_{\sigma_c.s} = \mathsf{true}
$$

$$
\begin{array}{rcl}
\mathsf{btids}(\mathbf{let}\ \Pi\ \mathbf{in}\ \hat{C}_1 \,\|\, \ldots \,\|\, \hat{C}_n, (\sigma_c, \sigma_o, \mathcal{K})) & \stackrel{\text{def}}{=} & (\{\mathrm{t} \mid \mathcal{K}(\mathrm{t}) = \circ \wedge \neg(\sigma_c \models \mathsf{en}(\hat{C}_{\mathrm{t}}))\}, \\
 & & \{\mathrm{t} \mid \mathcal{K}(\mathrm{t}) \neq \circ \wedge \neg((\sigma_o, \mathcal{K}(\mathrm{t})) \models \mathsf{en}(\hat{C}_{\mathrm{t}}))\})
\end{array}
$$

Fig. 3. States and event traces.

We make the following assumptions to simplify the technical setting. There are no regular function calls in either clients or objects. Therefore $x := f(E)$ can only be executed in client code to call object methods, and **return** E always returns from object methods to clients. Each object method takes only one argument and each method body ends with a **return** command. Object methods never execute the **print**(E) command and therefore do *not* generate external events. The command C in **await**$(B)\{C\}$ cannot contain **await**, **print**, and method calls and returns. It cannot contain loops either so that it always terminates.

Operational semantics. The operational semantics rules shown in Fig. 4 consist of three parts, including state transitions made by the whole program, by individual threads, and by clients or object methods, respectively. We define program states $\mathcal{S}$ in Fig. 3, where we use two sets of notations to represent states at the concrete and the abstract levels respectively when we study refinement. To ensure that the client code does not touch the object data, in $\mathcal{S}$ we separate the data accessed by clients (σ_c) and by object methods (σ_o). $\mathcal{S}$ also contains a *thread pool* $\mathcal{K}$ mapping thread IDs t to the corresponding method *call stacks* κ. Recall that the only function call allowed in the language is the method invocation made by a client and there are no nested function calls, therefore each κ is either empty ($\circ$, which means the thread is executing the *client* code), or contains only *one* stack frame (s_l, x, C), where s_l is the local store for the local variables of the method, x is the (client) variable recording the return value, and C is the continuation (the remaining client code to be executed after the return of the method).

Figure 4(a) shows that the execution of the program W follows the non-deterministic interleaving semantics, which is defined based on thread transitions defined in Fig. 4(b). Each transition over program configurations is labelled with a program event ι, a triple in the form of (e, Δ_c, Δ_o). The event e is generated by thread transitions. As defined in Fig. 3, (t, f, n) records the invocation of the method f with the argument n in the thread t, and $(\mathrm{t}, \mathbf{ret}, n)$ is for a method return with the return value n. $(\mathrm{t}, \mathbf{obj})$ and $(\mathrm{t}, \mathbf{clt})$ record a regular object step and a regular client step respectively, while $(\mathrm{t}, \mathbf{obj}, \mathbf{abort})$ and $(\mathrm{t}, \mathbf{clt}, \mathbf{abort})$ are for aborting of the thread in the object and client code

$$\frac{(\hat{C}_i,(\sigma_c,\sigma_o,\mathcal{K}(i))) \xrightarrow{e}_{i,\Pi} (\hat{C}'_i,(\sigma'_c,\sigma'_o,\kappa')) \qquad \mathcal{K}' = \mathcal{K}\{i \rightsquigarrow \kappa'\} \quad \text{btids}(\textbf{let}\ \Pi\ \textbf{in}\ \hat{C}_1 \,\|\, \ldots \hat{C}'_i \ldots \,\|\, \hat{C}_n,(\sigma'_c,\sigma'_o,\mathcal{K}')) = (\Delta_c,\Delta_o)}{(\textbf{let}\ \Pi\ \textbf{in}\ \hat{C}_1 \,\|\, \ldots \hat{C}_i \ldots \,\|\, \hat{C}_n,(\sigma_c,\sigma_o,\mathcal{K})) \xmapsto{(e,\Delta_c,\Delta_o)} (\textbf{let}\ \Pi\ \textbf{in}\ \hat{C}_1 \,\|\, \ldots \hat{C}'_i \ldots \,\|\, \hat{C}_n,(\sigma'_c,\sigma'_o,\mathcal{K}'))}$$

$$\frac{\hat{C}_i = \textbf{skip} \qquad \mathcal{K}(i) = \circ \qquad \hat{C}'_i = \textbf{end} \qquad e = (i,\textbf{term}) \quad \text{btids}(\textbf{let}\ \Pi\ \textbf{in}\ \hat{C}_1 \,\|\, \ldots \hat{C}_i \ldots \,\|\, \hat{C}_n,(\sigma_c,\sigma_o,\mathcal{K})) = (\Delta_c,\Delta_o)}{(\textbf{let}\ \Pi\ \textbf{in}\ \hat{C}_1 \,\|\, \ldots \hat{C}_i \ldots \,\|\, \hat{C}_n,(\sigma_c,\sigma_o,\mathcal{K})) \xmapsto{(e,\Delta_c,\Delta_o)} (\textbf{let}\ \Pi\ \textbf{in}\ \hat{C}_1 \,\|\, \ldots \hat{C}'_i \ldots \,\|\, \hat{C}_n,(\sigma_c,\sigma_o,\mathcal{K}))}$$

$$\frac{(\hat{C}_i,(\sigma_c,\sigma_o,\mathcal{K}(i))) \xrightarrow{e}_{i,\Pi} \textbf{abort}}{(\textbf{let}\ \Pi\ \textbf{in}\ \hat{C}_1 \,\|\, \ldots \hat{C}_i \ldots \,\|\, \hat{C}_n,(\sigma_c,\sigma_o,\mathcal{K})) \xmapsto{(e,\emptyset,\emptyset)} \textbf{abort}}$$

(a) program transitions

$$\frac{\Pi(f) = (\mathcal{P},y,C) \qquad \sigma_o \in \mathcal{P} \qquad [\![E]\!]_{s_c} = n \qquad x \in \mathit{dom}(s_c) \qquad \kappa = (\{y \rightsquigarrow n\},x,\mathbf{E}[\,\textbf{skip}\,])}{(\mathbf{E}[\,x := f(E)\,],((s_c,h_c),\sigma_o,\circ)) \xrightarrow{(\mathsf{t},f,n)}_{\mathsf{t},\Pi} (C,((s_c,h_c),\sigma_o,\kappa))}$$

$$\frac{f \notin \mathit{dom}(\Pi) \quad \text{or} \quad \sigma_o \notin \Pi(f).\mathcal{P} \quad \text{or} \quad [\![E]\!]_{s_c}\ \mathit{undefined} \quad \text{or} \quad x \notin \mathit{dom}(s_c)}{(\mathbf{E}[\,x := f(E)\,],((s_c,h_c),\sigma_o,\circ)) \xrightarrow{(\mathsf{t},\textbf{clt},\textbf{abort})}_{\mathsf{t},\Pi} \textbf{abort}}$$

$$\frac{\kappa = (s_l,x,C) \qquad [\![E]\!]_{s_l} = n \qquad s'_c = s_c\{x \rightsquigarrow n\}}{(\mathbf{E}[\,\textbf{return}\ E\,],((s_c,h_c),\sigma_o,\kappa)) \xrightarrow{(\mathsf{t},\textbf{ret},n)}_{\mathsf{t},\Pi} (C,((s'_c,h_c),\sigma_o,\circ))}$$

$$\frac{\kappa = (s_l,x,C) \qquad [\![E]\!]_{s_l}\ \mathit{undefined}}{(\mathbf{E}[\,\textbf{return}\ E\,],((s_c,h_c),\sigma_o,\kappa)) \xrightarrow{(\mathsf{t},\textbf{obj},\textbf{abort})}_{\mathsf{t},\Pi} \textbf{abort}}$$

$$\frac{[\![E]\!]_{s_c} = n}{(\mathbf{E}[\,\textbf{print}(E)\,],((s_c,h_c),\sigma_o,\circ)) \xrightarrow{(\mathsf{t},\textbf{out},n)}_{\mathsf{t},\Pi} (\mathbf{E}[\,\textbf{skip}\,],((s_c,h_c),\sigma_o,\circ))}$$

$$\frac{(C,(s_o \uplus s_l,h_o)) \mathrel{-\!\!\triangleright}_{\mathsf{t}} (C',(s'_o \uplus s'_l,h'_o)) \qquad \mathit{dom}(s_l) = \mathit{dom}(s'_l)}{(C,(\sigma_c,(s_o,h_o),(s_l,x,C_1))) \xrightarrow{(\mathsf{t},\textbf{obj})}_{\mathsf{t},\Pi} (C',(\sigma_c,(s'_o,h'_o),(s'_l,x,C_1)))} \qquad \frac{(C,\sigma_c) \mathrel{-\!\!\triangleright}_{\mathsf{t}} (C',\sigma'_c)}{(C,(\sigma_c,\sigma_o,\circ)) \xrightarrow{(\mathsf{t},\textbf{clt})}_{\mathsf{t},\Pi} (C',(\sigma'_c,\sigma_o,\circ))}$$

$$\frac{(C,(s_o \uplus s_l,h_o)) \mathrel{-\!\!\triangleright}_{\mathsf{t}} \textbf{abort}}{(C,(\sigma_c,(s_o,h_o),(s_l,x,C_1))) \xrightarrow{(\mathsf{t},\textbf{obj},\textbf{abort})}_{\mathsf{t},\Pi} \textbf{abort}} \qquad \frac{(C,\sigma_c) \mathrel{-\!\!\triangleright}_{\mathsf{t}} \textbf{abort}}{(C,(\sigma_c,\sigma_o,\circ)) \xrightarrow{(\mathsf{t},\textbf{clt},\textbf{abort})}_{\mathsf{t},\Pi} \textbf{abort}}$$

(b) thread transitions

$$\frac{[\![B]\!]_s = \textbf{true} \qquad (C,(s,h)) \mathrel{-\!\!\triangleright}^*_{\mathsf{t}} (\textbf{skip},(s',h'))}{(\mathbf{E}[\,\textbf{await}(B)\{C\}\,],(s,h)) \mathrel{-\!\!\triangleright}_{\mathsf{t}} (\mathbf{E}[\,\textbf{skip}\,],(s',h'))} \qquad \frac{[\![B]\!]_s = \textbf{true} \qquad (C,(s,h)) \mathrel{-\!\!\triangleright}^*_{\mathsf{t}} \textbf{abort}}{(\mathbf{E}[\,\textbf{await}(B)\{C\}\,],(s,h)) \mathrel{-\!\!\triangleright}_{\mathsf{t}} \textbf{abort}}$$

(c) local thread transitions

Fig. 4. Selected operational semantics rules.

respectively. The output event $(\mathsf{t},\textbf{out},n)$ is generated by the $\textbf{print}(E)$ command. $(\mathsf{t},\textbf{term})$ records the termination of the thread t. We also introduce a special event $(\textbf{spawn},n)$, which is inserted at the beginning of each event trace to record the creation of n threads at the beginning of the whole program execution. Its use is shown in Sec. 5. An *event trace* $\mathcal{E}$ is a (possibly infinite) sequence of events, and a *program trace* T is a (possibly infinite) sequence of labels ι.

The sets Δ_c and Δ_o in the label record the IDs of threads that are blocked in the client code and object methods respectively. They are generated by the function btids defined in Fig. 3. Recall that a thread t is executing the client code if its call stack is empty, i.e., $\mathcal{K}(\mathsf{t}) = \circ$. We also define $\mathsf{en}(\hat{C})$ as the enabling condition for $\hat{C}$, which ensures that $\hat{C}$ can execute at least one step unless it has terminated. Here the execution context $\mathbf{E}$ defines the position of the command to be executed next.

The second rule in Fig. 4(a) shows that **end** is used as a flag marking the termination of a thread. A termination event $(\mathsf{t}, \mathbf{term})$ is generated correspondingly.

The first two rules in Fig. 4(b) show that method calls can only be executed in the client code (i.e., when the stack κ is empty), and it is the clients' responsibility to ensure that the precondition $\mathcal{P}$ (defined in Fig. 3) of the method holds over the object data. If $\mathcal{P}$ does not hold, the method invocation step aborts. Similarly, as shown in the subsequent rules, the **return** command can only be executed in the object method, and the **print** command can only be in the client code. Other commands can be executed either in the client or in the object, and the transitions are made over σ_c and σ_o respectively. In Fig. 4(c) we show the operational semantics for $\mathbf{await}(B)\{C\}$. Note that there is no transition rule when B is false, which means that the thread is blocked. Transition rules of other commands are standard and omitted here.

More discussions about partial methods. There are actually two reasons that make a method partial. The first is due to non-termination when the method is called under certain conditions. The second is due to abnormal termination, i.e., the method aborts or terminates with incorrect states or return values. Since the goal of this work is to study progress, the paper focuses on the first kind of partial methods. In our language, we specify the two kinds of partial methods differently. For the first kind, we use the enabling condition B in $\mathbf{await}(B)\{C\}$ to specify when the method should *not* be blocked. For the second kind, we use the annotated precondition $\mathcal{P}$ to specify the condition needed for the method to execute safely and to generate correct results. For instance, although the lock's release method `L_REL` in specification (2.3) always terminates, it needs an annotated precondition `l=cid` to prevent client threads not owning the lock from releasing it.

4 LINEARIZABILITY AND BASIC CONTEXTUAL REFINEMENT

In this section we formally define linearizability [Herlihy and Wing 1990] of an object Π with respect to its abstract specification Γ. As explained in Sec. 2.2, Γ is an *atomic partial specification* for Π. It has the same syntax with Π (see Fig. 2), but each method body in Γ is always an await block $\mathbf{await}(B)\{C\}$ (followed by a **return** E command). We also assume that the methods in Γ are safe, i.e., they never abort.

History events, externally observable events, and traces. We call (t, f, n), $(\mathsf{t}, \mathbf{ret}, n)$ and $(\mathsf{t}, \mathbf{obj}, \mathbf{abort})$ *history* events, and $(\mathsf{t}, \mathbf{out}, n)$, $(\mathsf{t}, \mathbf{obj}, \mathbf{abort})$ and $(\mathsf{t}, \mathbf{clt}, \mathbf{abort})$ *externally observable* events. In Fig. 5 we define $\mathcal{T}[\![W, \mathcal{S}]\!]$ as the *prefix closed* set of finite traces T generated during the execution of $(W, \mathcal{S})$. $\mathcal{H}[\![W, \mathcal{S}]\!]$ contains the set of histories projected from traces in $\mathcal{T}[\![W, \mathcal{S}]\!]$. Here $\mathsf{get\_hist}(T)$ returns a subsequence $\mathcal{E}$ consisting of the history events projected from the corresponding labels in T. Similarly $\mathcal{O}[\![W, \mathcal{S}]\!]$ contains the set of externally observable event traces projected from traces in $\mathcal{T}[\![W, \mathcal{S}]\!]$, where $\mathsf{get\_obsv}(T)$ is a subsequence $\mathcal{E}$ consisting of externally observable events only.

As defined below, an event trace $\mathcal{E}$ is linearizable with respect to $\mathcal{E}'$, i.e., $\mathcal{E} \preceq^{\mathsf{lin}} \mathcal{E}'$, if they have the same sub-trace when projected over individual threads (projection represented as $\mathcal{E}|_{\mathsf{t}}$), and $\mathcal{E}$ is a permutation of $\mathcal{E}'$ but preserves the order of non-overlapping method calls in $\mathcal{E}'$. Here we use $\mathsf{is\_inv}(e)$ (or $\mathsf{is\_ret}(e_2)$) to represent that e is in the form of (t, f, n) (or $(\mathsf{t}, \mathbf{ret}, n)$).

$$\mathcal{T}[\![W,\mathcal{S}]\!] \overset{\text{def}}{=} \{T \mid \exists W',\mathcal{S}'.\ (W,\mathcal{S}) \xrightarrow{T}{}^{*} (W',\mathcal{S}') \vee (W,\mathcal{S}) \xrightarrow{T}{}^{*} \textbf{abort}\}$$
$$\mathcal{H}[\![W,\mathcal{S}]\!] \overset{\text{def}}{=} \{\mathcal{E} \mid \exists T.\ \mathcal{E} = \text{get\_hist}(T) \wedge T \in \mathcal{T}[\![W,\mathcal{S}]\!]\ \}$$
$$\mathcal{O}[\![W,\mathcal{S}]\!] \overset{\text{def}}{=} \{\mathcal{E} \mid \exists T.\ \mathcal{E} = \text{get\_obsv}(T) \wedge T \in \mathcal{T}[\![W,\mathcal{S}]\!]\ \}$$

$$\text{match}(e_1,e_2) \overset{\text{def}}{=} \text{is\_inv}(e_1) \wedge \text{is\_ret}(e_2) \wedge\ (\text{tid}(e_1) = \text{tid}(e_2))$$
$$\frac{}{\text{seq}(\epsilon)} \qquad \frac{\text{is\_inv}(e)}{\text{seq}(e :: \epsilon)} \qquad \frac{\text{match}(e_1,e_2) \qquad \text{seq}(\mathcal{E})}{\text{seq}(e_1 :: e_2 :: \mathcal{E})} \qquad \frac{\forall \text{t}.\ \text{seq}(\mathcal{E}|_{\text{t}})}{\text{well\_formed}(\mathcal{E})}$$
$$\frac{\text{well\_formed}(\mathcal{E})}{\mathcal{E} \in \text{extensions}(\mathcal{E})} \qquad \frac{\mathcal{E}' \in \text{extensions}(\mathcal{E}) \quad \text{is\_ret}(e) \quad \text{well\_formed}(\mathcal{E}'{+}{+}[e])}{\mathcal{E}'{+}{+}[e] \in \text{extensions}(\mathcal{E})}$$
$$\text{truncate}(\epsilon) \overset{\text{def}}{=} \epsilon \qquad \text{truncate}(e{::}\mathcal{E}) \overset{\text{def}}{=} \begin{cases} e{::}\text{truncate}(\mathcal{E}) & \text{if is\_ret}(e) \text{ or } \exists i.\ \text{match}(e,\mathcal{E}(i)) \\ \text{truncate}(\mathcal{E}) & \text{otherwise} \end{cases}$$
$$\text{completions}(\mathcal{E}) \overset{\text{def}}{=} \{\text{truncate}(\mathcal{E}') \mid \mathcal{E}' \in \text{extensions}(\mathcal{E})\}$$

$$\circledcirc \overset{\text{def}}{=} \{\text{t}_1 \leadsto \circ, \ldots, \text{t}_n \leadsto \circ\}$$
$$\Gamma \triangleright (\Sigma,\mathcal{E}) \text{ iff } \exists n, C_1,\ldots,C_n,\sigma_c.\ (\mathcal{E} \in \mathcal{H}[\![(\textbf{let}\ \Gamma\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n),(\sigma_c,\Sigma,\circledcirc)]\!]) \wedge \text{seq}(\mathcal{E})$$

Fig. 5. Auxiliary definitions for linearizability.

Definition 4.1 (Linearizable Histories). $\mathcal{E} \preceq^{\text{lin}} \mathcal{E}'$ iff both the following hold.

(1) $\forall \text{t}.\ \mathcal{E}|_{\text{t}} = \mathcal{E}'|_{\text{t}}$.
(2) There exists a bijection $\pi : \{1,\ldots,|\mathcal{E}|\} \to \{1,\ldots,|\mathcal{E}'|\}$ such that $\forall i.\ \mathcal{E}(i) = \mathcal{E}'(\pi(i))$ and
$$\forall i,j.\ i < j \wedge \text{is\_ret}(\mathcal{E}(i)) \wedge \text{is\_inv}(\mathcal{E}(j)) \implies \pi(i) < \pi(j)\,.$$

Definition 4.2 says Π is linearizable with respect to Γ and the state abstraction function φ (see Fig. 3) if, for any trace $\mathcal{E}$ generated by Π with the initial object data σ, the corresponding complete trace $\mathcal{E}_c$ is always linearizable with some *sequential* trace $\mathcal{E}'$ generated by Γ with initial object data Σ such that $\varphi(\sigma) = \Sigma$. Some of the key notations are defined in Fig. 5. We use $\circledcirc$ to represent the initial thread pool where each thread has an empty call stack. completions($\mathcal{E}$) appends matching return events for some pending invocations in $\mathcal{E}$, and discards the other pending invocations, so that in the resulting trace every invocation has a matching return. We use ++ for list concatenation, and $[e_1,\ldots,e_n]$ for a list consisting of a sequence of elements. We use tid(e) for the thread ID in e.

Definition 4.2 (Linearizability of Objects). The object implementation Π is linearizable with respect to Γ, written as $\Pi \preceq^{\text{lin}}_{\varphi} \Gamma$, iff
$$\forall n, C_1,\ldots,C_n,\sigma_c,\sigma,\Sigma,\mathcal{E}.\ \mathcal{E} \in \mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n),(\sigma_c,\sigma,\circledcirc)]\!] \wedge (\varphi(\sigma) = \Sigma)$$
$$\implies \exists \mathcal{E}_c,\mathcal{E}'.\ (\mathcal{E}_c \in \text{completions}(\mathcal{E})) \wedge (\Gamma \triangleright (\Sigma,\mathcal{E}')) \wedge (\mathcal{E}_c \preceq^{\text{lin}} \mathcal{E}')$$

Abstraction of linearizable objects. Filipović et al. [Filipović et al. 2009] has shown that linearizability is equivalent to a contextual refinement. As defined below, Π contextually refines Γ under the state abstraction function φ if, for any clients $C_1 \ldots C_n$ as the execution context, and for any initial object data related by φ, executing Π generates no more externally event traces than executing Γ. Theorem 4.4 shows the equivalence between linearizability and the contextual refinement.

Definition 4.3 (Basic Contextual Refinement). $\Pi \sqsubseteq^{\text{fin}}_{\varphi} \Gamma$ iff
$$\forall n, C_1,\ldots,C_n,\sigma_c,\sigma,\Sigma.\ (\varphi(\sigma) = \Sigma) \implies$$
$$\mathcal{O}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n),(\sigma_c,\sigma,\circledcirc)]\!] \subseteq \mathcal{O}[\![(\textbf{let}\ \Gamma\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n),(\sigma_c,\Sigma,\circledcirc)]\!]\,.$$

Theorem 4.4 (Basic Equivalence for Linearizability). $\Pi \preceq_{\varphi}^{\mathsf{lin}} \Gamma \iff \Pi \sqsubseteq_{\varphi}^{\mathsf{fin}} \Gamma$.

5 PROGRESS PROPERTIES

In this section we define the new progress properties, partial starvation-freedom (PSF) and partial deadlock-freedom (PDF), for objects with partial methods.

We first define the trace set $\mathcal{T}_{\omega}[\![W,\mathcal{S}]\!]$ in Fig. 6. It contains the (possibly infinite) *whole* execution traces T generated by $(W,\mathcal{S})$ but with a special label $((\mathbf{spawn}, |W|), \Delta_c, \Delta_o)$ inserted at the beginning. Here we use $|W|$ to represent the number of threads in W. The event $(\mathbf{spawn}, |W|)$ is used to define fairness, as explained below. Δ_c and Δ_o records the threads blocked in clients and object methods respectively (see the definition of btids in Fig. 3). At the beginning of an execution, Δ_o must be an empty set since no threads are in method calls. The whole execution trace T may be generated under three cases, i.e., the execution of $(W,\mathcal{S})$ diverges, aborts or gets stuck (terminates or is blocked). We write $(W,\mathcal{S}) \overset{T}{\longmapsto}{}^{\omega} \cdot$ for an infinite execution. In this case, the length of T must be infinite, written as $|T| = \omega$.

Strong and weak fairness. As defined in Fig. 6, a trace T is strongly fair, represented as $\mathsf{sfair}(T)$, if each thread either terminates, or is executed infinitely many times if it is *infinitely often* enabled (i-o-enabled). We know a thread is enabled if it is not in the blocked sets Δ_c and Δ_o. $T(j)$ represents the j-th element in the trace T. Similarly, $\mathsf{wfair}(T)$ says that T is a weakly fair trace. It requires that each thread either terminates, or is executed infinitely many times if it is *always* enabled after certain step on the trace (e-a-enabled).

Thread progress and program progress. We use $\mathsf{prog\text{-}t}(T)$ in Fig. 6 to say that every method call eventually terminates. It ensures that each individual thread calling a method eventually returns. $\mathsf{prog\text{-}p}(T)$ says that there is always at least one method call that terminates. It ensures that the whole program is making progress. Here $\mathsf{pend\_inv}(T)$ represents the set of method invocation events that do not have matching returns. $T(1..i)$ represents the prefix of T with length i.

Partial starvation-freedom (PSF) and partial deadlock-freedom (PDF). We want to define PSF as a generalization of starvation-freedom. We say an object Π is *partially starvation-free* if, under fair scheduling (with strong or weak fairness), each method call eventually returns (as required in starvation-freedom), unless it is eventually always disabled (i.e., it is not supposed to return in this particular execution context). In the latter case the non-termination is caused by inappropriate invocations of the methods in the client code and the object implementation should *not* be blamed.

Although the idea is intuitive, it is challenging to formalize it. This is because when we say a method is disabled we are thinking at an abstract level, where the abstract disabling condition cannot be syntactically inferred based on the low-level object implementation Π. For instance, the lock implementations in Fig. 1 use non-blocking commands only, so they are always enabled to execute one more step at this level, although we intend to say at a more abstract level that the `L_acq()` operation is disabled when the lock is unavailable.

To address this problem, we refer to the abstract object specification Γ when defining the progress of a concrete object Π. Recall that method specifications in Γ are in the form of $\mathbf{await}(B)\{C\}$, so we know that the method is disabled when B does not hold.

We formalize the idea as Def. 5.1. Under the scheduling fairness χ (where $\chi \in \{\mathsf{sfair}, \mathsf{wfair}\}$, as defined in Fig. 6), we say the object Π is PSF with respect to an abstract specification Γ and a state abstraction function φ, i.e., $\mathsf{PSF}_{\varphi,\Gamma}^{\chi}(\Pi)$, if any χ-fair trace T generated by $((\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma, \circledcirc))$ either aborts, or satisfies prog-t, or we could blame the client for the blocking of each pending invocation.

$$\begin{array}{l}\mathcal{T}_\omega[\![W,\mathcal{S}]\!] \overset{\text{def}}{=} \{((\mathbf{spawn},|W|),\Delta_c,\Delta_o)::T \mid \mathsf{btids}(W,\mathcal{S}) = (\Delta_c,\Delta_o) \wedge \\ \quad (((W,\mathcal{S}) \overset{T}{\longmapsto}{}^{\omega} \cdot) \vee ((W,\mathcal{S}) \overset{T}{\longmapsto}{}^{*} \mathbf{abort}) \vee \exists W',\mathcal{S}'.\ ((W,\mathcal{S}) \overset{T}{\longmapsto}{}^{*} (W',\mathcal{S}')) \wedge \neg(\exists\iota.\ (W',\mathcal{S}') \overset{\iota}{\longmapsto} \_)) \}\end{array}$$

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

$$\mathsf{evt}(\iota) \overset{\text{def}}{=} e \quad \text{if } \iota = (e,\Delta_c,\Delta_o) \qquad \mathsf{bset}(\iota) \overset{\text{def}}{=} \Delta_c \cup \Delta_o \quad \text{if } \iota = (e,\Delta_c,\Delta_o)$$

$$\mathsf{i\text{-}o\text{-}enabled}(\mathsf{t},T) \ \text{ iff } \ \forall i.\ \exists j \geq i.\ \mathsf{t} \notin \mathsf{bset}(T(j)) \qquad \text{“infinitely often”}$$
$$\mathsf{e\text{-}a\text{-}enabled}(\mathsf{t},T) \ \text{ iff } \ \exists i.\ \forall j \geq i.\ \mathsf{t} \notin \mathsf{bset}(T(j)) \qquad \text{“eventually always”}$$

$$\mathsf{sfair}(T) \ \text{ iff } \ |T|{=}\omega \implies \forall \mathsf{t} \in [1..\mathsf{tnum}(T)].\ \mathsf{evt}(\mathsf{last}(T|_{\mathsf{t}})) = (\mathsf{t},\mathbf{term}) \vee (\mathsf{i\text{-}o\text{-}enabled}(\mathsf{t},T) \Rightarrow |(T|_{\mathsf{t}})| = \omega)$$
$$\mathsf{wfair}(T) \ \text{ iff } \ |T|{=}\omega \implies \forall \mathsf{t} \in [1..\mathsf{tnum}(T)].\ \mathsf{evt}(\mathsf{last}(T|_{\mathsf{t}})) = (\mathsf{t},\mathbf{term}) \vee (\mathsf{e\text{-}a\text{-}enabled}(\mathsf{t},T) \Rightarrow |(T|_{\mathsf{t}})| = \omega)$$

$$\mathsf{pend\_inv}(T) \overset{\text{def}}{=} \{e \mid \exists i.\ e = \mathsf{evt}(T(i)) \wedge \mathsf{is\_inv}(e) \wedge \neg\exists j > i.\ \mathsf{match}(e,\mathsf{evt}(T(j)))\ \}$$
$$\mathsf{abt}(T) \ \text{ iff } \ \exists i.\ \mathsf{is\_abt}(\mathsf{evt}(T(i)))$$
$$\mathsf{prog\text{-}t}(T) \ \text{ iff } \ \mathsf{pend\_inv}(T) = \emptyset$$
$$\mathsf{prog\text{-}p}(T) \ \text{ iff } \ \forall i,e.\ e \in \mathsf{pend\_inv}(T(1..i)) \implies \exists j > i.\ \mathsf{is\_ret}(\mathsf{evt}(T(j)))$$

$$\mathsf{e\text{-}a\text{-}disabled}(\mathsf{t},T) \ \text{ iff } \ \exists i.\ \forall j \geq i, \iota = T(j).\ \mathsf{t} \in \mathsf{bset}(\iota) \qquad \text{“eventually always”}$$
$$\begin{array}{l}\mathsf{well\text{-}blocked}(T,(W_a,\mathcal{S}_a)) \ \text{ iff } \ \exists T_a.\ T_a \in \mathcal{T}_\omega[\![W_a,\mathcal{S}_a]\!] \wedge (\mathsf{get\_hist}(T) = \mathsf{get\_hist}(T_a)) \\ \qquad\qquad\qquad\qquad\qquad\qquad \wedge (\forall e.\ e \in \mathsf{pend\_inv}(T_a) \implies \mathsf{e\text{-}a\text{-}disabled}(\mathsf{tid}(e),T_a))\end{array}$$

$$\mathcal{O}_\chi[\![W,(\sigma_c,\sigma_o)]\!] \overset{\text{def}}{=} \{\mathcal{E} \mid \exists T.\ T \in \mathcal{T}_\omega[\![W,(\sigma_c,\sigma_o,\circledcirc)]\!] \wedge \chi(T) \wedge \mathsf{get\_obsv}(T) = \mathcal{E}\} \qquad \chi \in \{\mathsf{sfair},\mathsf{wfair}\}$$

Fig. 6. Fairness and progress.

In the last case, we must be able to find a trace T_a generated by the execution of the abstract object Γ (with the abstract object state Σ related to σ by φ) such that it has the *same* method invocation and return history with T, and every pending invocation in this abstract trace T_a is eventually always disabled. See the definition of well-blocked in Fig. 6 for the formal details.

Definition 5.1 (Partially Starvation-Free Objects). $\mathsf{PSF}^{\chi}_{\varphi,\Gamma}(\Pi)$ iff

$$\begin{array}{l}\forall n,C_1,\ldots,C_n,\sigma_c,\sigma,\Sigma,T.\ T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n),(\sigma_c,\sigma,\circledcirc)]\!] \wedge (\varphi(\sigma) = \Sigma) \wedge \chi(T) \\ \implies \mathsf{abt}(T) \vee \mathsf{prog\text{-}t}(T) \vee \mathsf{well\text{-}blocked}(T,((\mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n),(\sigma_c,\Sigma,\circledcirc)))\,.\end{array}$$

We also define PDF in Def. 5.2. It is similar to PSF, but requires prog-p instead of prog-t.

Definition 5.2 (Partially Deadlock-Free Objects). $\mathsf{PDF}^{\chi}_{\varphi,\Gamma}(\Pi)$ iff

$$\begin{array}{l}\forall n,C_1,\ldots,C_n,\sigma_c,\sigma,\Sigma,T.\ T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n),(\sigma_c,\sigma,\circledcirc)]\!] \wedge (\varphi(\sigma) = \Sigma) \wedge \chi(T) \\ \implies \mathsf{abt}(T) \vee \mathsf{prog\text{-}p}(T) \vee \mathsf{well\text{-}blocked}(T,((\mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n),(\sigma_c,\Sigma,\circledcirc)))\,.\end{array}$$

The above definitions consider the three factors that may affect the termination of a method call: the scheduling fairness χ, the object implementation Π which determines whether its traces satisfy prog-t or prog-p, and the execution context $C_1 \parallel \ldots \parallel C_n$ which may make inappropriate method invocations so that well-blocked holds. Consider the lock objects in Fig. 1(a) and (c) and the following client program (5.1). The initial value of the shared variable x is 0.

```
[_]ACQ; print(0); [_]REL;   ||  [_]ACQ; print(2);
x:=1;                       ||  while(x=1){                      (5.1)
[_]ACQ; print(1); [_]REL;   ||    [_]REL; [_]ACQ; print(3); }
```

The client can produce a trace satisfying prog-t when it uses the ticket lock. It first executes the left thread until termination and then executes the right thread. Then every method call terminates, printing out 0, 1, 2 and an infinite number of 3. Thus prog-t holds. When the test-and-set lock is used instead, the same client can produce a trace satisfying prog-p but not prog-t. In the execution, the second call to `L_acq` in the left thread never finishes. It prints out 0, 2 and an infinite number of 3, but not 1. Such an execution is not possible when the client uses the ticket lock, under fair scheduling. This shows how different object implementations affect termination of method calls. Note that neither of the two execution traces satisfies well-blocked, because every method call in the traces either terminates or is enabled infinitely often.

On the other hand, the client (5.1) can produce a well-blocked trace no matter it uses the ticket lock or the test-and-set lock. It executes the right thread first until termination and then executes the left thread. Then the first call to lock acquire of the left thread is always blocked, and only 2 is printed during the execution. The non-termination of the method call is caused by the particular execution context, in which the method call is not supposed to return, regardless of the object implementations. This is why the same well-blocked condition is used in both definitions of PSF and PDF for both strongly and weakly fair executions of the object implementation.

PSF (or PDF) and starvation-freedom (or deadlock-freedom) coincide if we require each **await** block in Γ is in the special form of $\mathbf{await}(\mathrm{true})\{C\}$ — Since the methods in Γ are always enabled, $\text{well-blocked}(T, ((\mathbf{let}\ \Gamma\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n), \mathcal{S}_a))$ now requires that there is no pending invocation in T. This is stronger than both $\text{prog-t}(T)$ and $\text{prog-p}(T)$. Therefore we can remove the disjunction branch about well-blocked in Defs. 5.1 and 5.2, resulting in definitions equivalent to starvation-freedom and deadlock-freedom respectively.

6 PROGRESS-AWARE ABSTRACTION OF OBJECTS

In this section we study the abstraction of linearizable and PSF (or PDF) objects. Similar to Theorem 4.4, we want theorems showing that linearizability along with PSF (or PDF) of an object Π is equivalent to a contextual refinement between Π and some abstract object Π', where Π' can be syntactically derived from the atomic specification Γ.

We first define the progress-aware contextual refinement for objects under different fairness χ of scheduling ($\chi \in \{\mathrm{sfair}, \mathrm{wfair}\}$). As Def. 6.1 shows, Π contextually refines Π' under the χ-fair scheduling if, in any execution context, Π generates no more externally observable event traces than Π'. The set of event traces $O_\chi[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma)]\!]$ is defined in Fig. 6, where each event trace $\mathcal{E}$ is extracted from the χ-fair trace T in $\mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma, \circledcirc)]\!]$. The refinement is *progress-aware* because we use the whole execution trace T here, from which we can tell whether the corresponding execution terminates or not.

Definition 6.1 (Progress-Aware Contextual Refinement).

$$\Pi \sqsubseteq_\varphi^\chi \Pi' \text{ iff } \forall n, C_1, \ldots, C_n, \sigma_c, \sigma, \Sigma.\ \varphi(\sigma) = \Sigma \implies O_\chi[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma)]\!] \subseteq O_\chi[\![(\mathbf{let}\ \Pi'\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \Sigma)]\!]$$

Wrappers for atomic specifications. As explained in Sec. 2, one of the major contributions of this paper is to define wrappers for atomic partial specifications Γ, which transform the method specification $\mathbf{await}(B)\{C\}$ in Γ into a (possibly non-atomic) abstract specification for each combination of progress (PSF or PDF) and fairness (sfair or wfair), as shown in Table 2.

$$\begin{array}{rcl}
\mathsf{wr}^{\chi}_{\mathsf{Prog}}(\Gamma)(f) & \stackrel{\text{def}}{=} & (\mathcal{P}, x, \mathsf{wr}^{\chi}_{\mathsf{Prog}}(\mathbf{await}(B)\{C\}); \mathbf{return}\ E) \\
& & \qquad\qquad \text{if } \Gamma(f) = (\mathcal{P}, x, \mathbf{await}(B)\{C\}; \mathbf{return}\ E) \\
\mathsf{wr}^{\mathsf{sfair}}_{\mathsf{PSF}}(\mathbf{await}(B)\{C\}) & \stackrel{\text{def}}{=} & \mathbf{await}(B)\{C\} \\
\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PSF}}(\mathbf{await}(B)\{C\}) & \stackrel{\text{def}}{=} & \texttt{listid} := \texttt{listid}{+}{+}[(\texttt{cid}, `B\text{'})]; \\
& & \mathbf{await}(B \land \texttt{cid} = \mathbf{enhd}(\texttt{listid}))\{C;\ \texttt{listid} := \texttt{listid} \backslash \texttt{cid};\} \\
\mathsf{wr}^{\mathsf{sfair}}_{\mathsf{PDF}}(\mathbf{await}(B)\{C\}) & \stackrel{\text{def}}{=} & \mathbf{while}\ (\mathsf{done})\{\}; \\
& & \mathbf{await}(B \land \neg\mathsf{done})\{C; \mathsf{done} := \mathsf{true};\};\ \ \mathsf{done} := \mathsf{false}; \\
& & \mathbf{while}\ (\mathsf{done})\{\}; \\
\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PDF}}(\mathbf{await}(B)\{C\}) & \stackrel{\text{def}}{=} & \mathbf{await}(B \land \neg\mathsf{done})\{C; \mathsf{done} := \mathsf{true};\};\ \ \mathsf{done} := \mathsf{false}; \\
& & \mathbf{await}(\neg\mathsf{done})\{\}
\end{array}$$

$$\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PSF}}(\varphi)(\sigma) \stackrel{\text{def}}{=} \begin{cases} \sigma' \uplus \{\texttt{listid} \rightsquigarrow \epsilon\} & \text{if } \varphi(\sigma) = \sigma' \\ \mathit{undefined} & \text{if } \sigma \notin \mathit{dom}(\varphi) \end{cases} \qquad \mathsf{wr}^{\mathsf{sfair}}_{\mathsf{PSF}}(\varphi) \stackrel{\text{def}}{=} \varphi$$

$$\mathsf{wr}^{\chi}_{\mathsf{PDF}}(\varphi)(\sigma) \stackrel{\text{def}}{=} \begin{cases} \sigma' \uplus \{\mathsf{done} \rightsquigarrow \mathbf{false}\} & \text{if } \varphi(\sigma) = \sigma' \\ \mathit{undefined} & \text{if } \sigma \notin \mathit{dom}(\varphi) \end{cases}$$

Fig. 7. Definition of wrappers.

Before introducing the definition of the wrappers in Fig. 7, we first show our abstraction theorem (Theorem 6.2) for linearizable and PSF (or PDF) objects. It establishes the equivalence between the progress-aware contextual refinement and linearizability with PSF (or PDF).

Theorem 6.2 (Abstraction Theorem). *Let* $\mathsf{Prog} \in \{\mathsf{PSF}, \mathsf{PDF}\}$ *and* $\chi \in \{\mathsf{sfair}, \mathsf{wfair}\}$*, then*

$$\Pi \preceq^{\mathsf{lin}}_{\varphi} \Gamma \land \mathsf{Prog}^{\chi}_{\varphi,\Gamma}(\Pi) \iff \Pi \sqsubseteq^{\chi}_{\widehat{\varphi}} \mathsf{wr}^{\chi}_{\mathsf{Prog}}(\Gamma)\ ,$$

where $\widehat{\varphi} = \mathsf{wr}^{\chi}_{\mathsf{Prog}}(\varphi)$*, and the wrappers for* Γ *and* φ *are defined in Fig. 7. We also assume that the variables* `listid` *and* done *introduced in the wrapper code are fresh, i.e.,* $\texttt{listid}, \mathsf{done} \notin \mathsf{FV}(\{\Pi, \Gamma, \varphi\})$.

To prove the theorem, we define compositional operational semantics that can generate separate traces for objects and clients, and build simulations using the object semantics. Detailed proofs are given in the TR [Liang and Feng 2017].

Next we introduce the definition of the wrappers in detail. The wrapper $\mathsf{wr}^{\mathsf{sfair}}_{\mathsf{PSF}}$ is simply an identity function. It maps the atomic specification $\mathbf{await}(B)\{C\}$ to itself. This is because under *strongly fair scheduling* $\mathbf{await}(B)\{C\}$ will eventually be executed unless it is eventually always disabled. This is exactly what we need for PSF of linearizable objects, which requires that the invocation of each method eventually returns, unless the corresponding high-level atomic operation $\mathbf{await}(B)\{C\}$ is eventually always disabled (as specified by well-blocked in Fig. 6).

Wrapper for PSF under weakly fair scheduling. Under weakly fair scheduling, however, we cannot guarantee that $\mathbf{await}(B)\{C\}$ is eventually executed even if B holds infinitely often. Therefore it alone cannot satisfy PSF. That's why we define $\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PSF}}(\mathbf{await}(B)\{C\})$, which guarantees that the atomic operation is eventually executed if B holds infinitely often. We introduce a blocking queue `listid` in the object state, which is a sequence of (t, 'B') pairs, showing that the thread t requests to execute an atomic operation with the enabling condition B. Note that the enabling condition B is recorded *syntactically* in `listid`, represented as 'B'. The operator $\mathbf{enhd}(\texttt{listid})$ returns the first thread on the list whose enabling condition is true. It evaluates the syntactic enabling conditions 'B' recorded in `listid` on the fly. Note that different pairs in `listid` may have different enabling conditions B. In the code generated by $\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PSF}}(\textbf{await}(B)\{C\})$, we first append the current thread ID and the enabling condition 'B' at the end of the list. In the subsequent command the thread waits until both B holds and $\mathtt{cid} = \textbf{enhd}(\mathtt{listid})$[1]. Then it atomically executes C and deletes the current thread in the queue.

This wrapper guarantees that C is eventually executed when B becomes infinitely often true because we know $B \wedge \mathtt{cid} = \textbf{enhd}(\mathtt{listid})$ will be *eventually always* true, and then the weakly fair scheduling guarantees the execution of C. This is because, whenever B becomes true, either $\mathtt{cid} = \textbf{enhd}(\mathtt{listid})$ holds or there is a pair (t', B') such that $B' \wedge \mathsf{t}' = \textbf{enhd}(\mathtt{listid})$ holds. In the first case, other threads trying to execute the object methods must be blocked at the **await** command. Therefore B cannot be changed to false by other threads. Therefore $B \wedge \mathtt{cid} = \textbf{enhd}(\mathtt{listid})$ is always true until the current thread executes the atomic block. In the second case t' must be able to finish its method, following the same argument above. Therefore there will be one less thread waiting in front of the current thread `cid`. Since B becomes true infinitely often, we eventually reach the first case.

As a result, the wrapper does not terminate in a weakly fair execution only if B is eventually always false. In that case the execution trace is well-blocked (see Fig. 6), still satisfying PSF.

One may argue that the abstraction generated by the wrapper is not very useful because it may not be much simpler than the object implementation. For instance, if we consider the `acquire` method of locks, the abstraction is almost the same as queue locks or ticket locks. But we want to emphasize that our wrapper is a general one that works for any object method implementation with an atomic specification in the form of $\textbf{await}(B)\{C\}$. Therefore we know the method's progress-aware abstraction can always be in this form, no matter how complex its implementation is.

Wrapper for PDF under weakly fair scheduling. For the right column in Table 2, we first introduce the wrapper at the bottom right corner. The definition of PDF says a method can be non-terminating if (1) it is eventually always disabled, as specified by well-blocked (see Fig. 6); or (2) there are always other method calls terminating, as specified by prog-p. Note that the second condition allows the method to be non-terminating even if it is eventually always enabled under weakly fair scheduling. As an example, the Treiber stack with a partial pop in Fig. 8 demonstrates one such scenario. The pop method is blocked when the stack is empty. It is linearizable with respect to the following specification

$$\textbf{await}(\mathsf{S} \neq \mathtt{nil})\{\mathsf{tmp} := \mathsf{head}(\mathsf{S}); \mathsf{S} := \mathtt{tail}(\mathsf{S});\}; \ \ \textbf{return}\ \mathsf{tmp}; \tag{6.1}$$

where S is the abstraction of the stack and tmp is a thread-local temporary variable.

In the following execution context (6.2),

```
pop();   ||   while(true){ push(0); }          (6.2)
```

the call of the concrete method pop may never terminate because its **cas** command may always fail, although the enabling condition at the abstract level ($\mathsf{S} \neq \mathtt{nil}$) is eventually always true. However, if we replace the method implementation with the specification (6.1), pop must terminate under weakly fair scheduling. This shows that the concrete implementation cannot contextually refine this simple specification (6.1).

Our first attempt to address this problem is to introduce a new object variable done (initialized to `false`), and let the wrapper $\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PDF}}$ transform $\textbf{await}(B)\{C\}$ into:

$$\textbf{await}(B \wedge \neg\mathsf{done})\{C;\ \mathsf{done} := \mathtt{true}\}; \quad \mathsf{done} := \mathtt{false}; \tag{6.3}$$

[1] Actually the conjunct B in the **await** condition in the wrapper could be omitted, because B must be true when $\mathtt{cid} = \textbf{enhd}(\mathtt{listid})$ holds.

```
initialize(){  Top := null; }

push(v){
  local x, b, t;
  b := false;
  x := cons(v, null);
  while (!b) {
    t := Top;
    x.next := t;
    b := cas(&Top, t, x);
  }
}
```

```
pop(){
  local x, b, t, v;
  b := false;
  while (!b) {
    t := Top;
    if (t != null) {
      v := t.data;
      x := t.next;
      b := cas(&Top, t, x);
    }
  }
  return v;
}
```

```
initialize'(){  initialize(); done := false;}

push'(v){ push(v); DLY_NOOP}        pop'(v){ tmp := pop(); DLY_NOOP; return tmp}
```

DLY_NOOP $\stackrel{\text{def}}{=}$ **await**(¬done){done := true}; done := false;

```
                 ║ push'(1);
                 ║ push'(2);
r0 := pop'();    ║ r1 := pop'();
print(r0);       ║ print(r1);
                 ║ while(true}{ push'(0) };
```

Fig. 8. Treiber stacks with partial pops.

Therefore the resulting **await** command may not be executed even if B is always true, because done can be set to `true` infinitely often when other threads finish the atomic block. Also note done is reset to `false` at the end of each **await** command, therefore the condition ¬done cannot always disable the **await** command, which may cause deadlock. As a result, there is always some thread that can finish the wrapper (i.e., prog-p holds) unless the B-s of all the pending invocations are eventually always false (i.e., well-blocked holds), thus PDF holds.

However, this is not the end of the story. If the code (6.3) fails to terminate, C must not be executed and no effects (over the object data) are generated. However, it is possible for PDF methods to finish C and make the effects visible to other threads but fail to terminate. As an example we define the `push'` and `pop'` methods in Fig. 8 as a new implementation of the Treiber stack. They call the push and pop methods respectively and then execute the code snippet DLY_NOOP before they return. DLY_NOOP simply waits until done becomes `false` and then atomically sets it to `true`, and finally resets it to `false`. The only purpose of DLY_NOOP is to allow the methods to be delayed by other threads or to delay others.

Then we consider the client code shown at the bottom of Fig. 8. Under weakly fair scheduling it is possible that the call of `pop'()` by the left thread never terminates but the thread on the right prints out 1. That is, although the `pop'()` on the left does not terminate, it does generate effects over the stack and the effects happen before the `pop'()` on the right. Such an external event trace cannot be generated if we replace the concrete `push'()` and `pop'()` methods with the abstract method code generated using the wrapper (6.3) defined above. Thus the contextual refinement between the concrete code and the wrapped specification does not hold.

Our solution is to append an **await** command at the end of (6.3), so that the resulting code $\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PDF}}(\mathbf{await}(B)\{C\})$ (see Fig. 7) may finish C but still be blocked at the end.

Wrapper for PDF under strongly fair scheduling. Much of the effort to define $\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PDF}}(\mathbf{await}(B)\{C\})$ is to allow the resulting code to be non-terminating even if B is eventually always true. We need to do the same to define $\mathsf{wr}^{\mathsf{sfair}}_{\mathsf{PDF}}(\mathbf{await}(B)\{C\})$, but it is more challenging with *strongly fair scheduling* because $\mathbf{await}(\neg\mathsf{done})\{\}$ cannot be blocked under strongly fair scheduling if done is infinitely often true. Therefore we use **while**-loops in $\mathsf{wr}^{\mathsf{sfair}}_{\mathsf{PDF}}(\mathbf{await}(B)\{C\})$ to allow the method to be delayed when done is infinitely often true[2]. Note that **while** (done){} terminates when done is false.

Wrappers for the state abstraction function. Since the program transformations by the wrappers introduce new object variables such as `listid` and done, we need to change the state abstraction function φ accordingly, which is defined as $\mathsf{wr}^{\chi}_{\mathsf{Prog}}(\varphi)$ in Fig. 7 ($\chi \in \{\mathsf{sfair}, \mathsf{wfair}\}$ and $\mathsf{Prog} \in \{\mathsf{PSF}, \mathsf{PDF}\}$).

More discussions. There could be different ways to define the wrappers to validate the Abstraction Theorem 6.2. We do not intend to claim that our definitions are the simplest ones (and it is unclear how to formally compare the complexity of different wrappers), but we would like to point out that, although some of the wrappers look complex, the complexity is partly due to the effort to have general wrappers that work for any atomic specifications in the form of $\mathbf{await}(B)\{C\}$. It is possible to have simpler wrappers for specific objects. For instance, the lock specification Γ in (2.3) defined in Sec. 2.2 can already serve as an abstraction for the test-and-set lock object Π_{TAS} (which is a PDF lock) under weakly fair scheduling, i.e., $\Pi_{\mathrm{TAS}} \sqsubseteq^{\mathsf{wfair}}_{\varphi} \Gamma$ holds.

7 PROGRAM LOGIC

We extend the program logic LiLi [Liang and Feng 2016] to verify progress properties of concurrent objects with partial methods. LiLi is a rely-guarantee style program logic to verify linearizability and starvation-freedom/deadlock-freedom of concurrent objects. It establishes progress-aware contextual refinements between concrete object implementations and abstract (total) specifications.

The key ideas of LiLi to verify progress are the following:

- A thread can be blocked, relying on the actions of other threads (i.e., its environment) to make progress. To ensure it eventually progresses, we must guarantee that the environment actions that the thread waits for eventually occur.
- To avoid circularity in rely-guarantee reasoning, each thread specifies a set of *definite actions* $\mathcal{D}$, which are state transitions specified in the form of $P \leadsto Q$. The thread guarantees that, whenever a definite action $P \leadsto Q$ is "enabled" (i.e., the assertion P holds), the transition must occur so that Q eventually holds, regardless of the environment behaviors.
- A blocked thread must wait for a set of definite actions of other threads, and the size of the set must be decreasing (so that the thread is eventually unblocked).
- A thread may delay the progress of others, i.e., to make other threads to execute more steps than they need when executed in isolation. To ensure deadlock-freedom, LiLi disallows a thread to be delayed infinitely often without whole system progress. This is achieved by using tokens as resources and each delaying action must consume a token.

These ideas to reason about blocking and delay are general enough for verifying objects with partial methods, but we have to first generalize LiLi in the following two aspects:

[2]Actually the conjunct $\neg$done in the **await** condition in the wrapper could be removed, because the loop **while** (done){} before the **await** block can already produce the non-terminating behaviors when other threads finish the method infinitely often (i.e., done is infinitely often true). Here we keep the conjunct $\neg$done to make the wrapper more intuitive.

$$
\begin{array}{rrcl}
(\mathit{RelAssn}) & P,Q,J & ::= & B \mid \mathsf{emp} \mid E \mapsto E \mid E \Mapsto E \mid \llbracket p \rrbracket \mid P * Q \mid P \wedge Q \mid P \vee Q \mid \ldots \\
(\mathit{FullAssn}) & p,q & ::= & P \mid \mathsf{arem}(C) \mid \lozenge(E) \mid \blacklozenge(E_k,\ldots,E_1) \mid p * q \mid p \wedge q \mid \ldots \\
(\mathit{RelAct}) & R,G & ::= & P \ltimes_k Q \mid [P] \mid \lfloor G \rfloor_0 \mid \mathcal{D} \mid G \wedge G \mid G \vee G \mid \ldots \\
(\mathit{DAct}) & \mathcal{D} & ::= & P \leadsto Q \mid \forall x.\, \mathcal{D} \mid \mathcal{D} \wedge \mathcal{D}
\end{array}
$$

$$\mathfrak{S} ::= (\sigma, \Sigma) \qquad u ::= (n_k,\ldots,n_1) \qquad w \in \mathit{Nat}$$

$$
\begin{array}{rcl}
\mathsf{Enabled}(P \leadsto Q) & \overset{\text{def}}{=} & P \\
\mathsf{Enabled}(\forall x.\, \mathcal{D}) & \overset{\text{def}}{=} & \exists x.\, \mathsf{Enabled}(\mathcal{D}) \\
\mathsf{Enabled}(\mathcal{D}_1 \wedge \mathcal{D}_2) & \overset{\text{def}}{=} & \mathsf{Enabled}(\mathcal{D}_1) \vee \mathsf{Enabled}(\mathcal{D}_2)
\end{array}
\qquad
\begin{array}{l}
\langle \mathcal{D} \rangle \overset{\text{def}}{=} \mathcal{D} \wedge (\mathsf{Enabled}(\mathcal{D}) \ltimes \mathsf{true}) \\
\mathcal{D}' \leqslant \mathcal{D} \ \text{iff} \ (\mathsf{Enabled}(\mathcal{D}') \Rightarrow \mathsf{Enabled}(\mathcal{D})) \\
\qquad\qquad \wedge (\mathcal{D} \Rightarrow \mathcal{D}')
\end{array}
$$

Fig. 9. Assertions and models.

- LiLi does not have **await** commands. There **while**-loops are the only commands that affect progress. Now we have to reason about **await** in object code, which may affect progress as well. It is interesting to see that **await** can be reasoned about similarly as **while**-loops.
- We also have **await**$(B)\{C\}$ as partial specifications. Since we want *termination-preserving* refinement, we do not have to guarantee progress of the concrete object methods when the partial specification is disabled.

As an extension of LiLi, our logic borrows LiLi's key ideas and most of the logic rules. Due to the space limit, the full details of the logic are given in the TR [Liang and Feng 2017]. Below we only show the major extensions.

7.1 Assertions

The assertions, shown in Fig. 9, are the same as those in LiLi. P and Q are assertions over relational states $\mathfrak{S}$, which are pairs of concrete and abstract object states (see Fig. 9). We use relational assertions because our logic establishes refinement between the concrete object implementation and the abstract specification. As in separation logic, we use $E \mapsto E$ and $E \Mapsto E$ to specify memory cells at the concrete and abstract level respectively. emp specifies empty states, and the separating conjunction $P * Q$ specifies two disjoint parts of $\mathfrak{S}$, which satisfy P and Q respectively.

The full assertions p and q specify triples in the form of $(\mathfrak{S}, (u, w), C)$. In addition to the relational state $\mathfrak{S}$, the assertions also describe the numbers (u, w) of available tokens, and the high-level (abstract) method code C that remains to be refined by the low-level (concrete) code. The assertion $\mathsf{arem}(C)$ specifies such high-level code in the triple.

Tokens and multi-level delaying actions. As explained above, LiLi uses tokens as resources to prevent infinitely many execution steps and infinitely many delaying actions. It requires that each round of a **while**-loop consumes a $\lozenge$-token, and each delaying action consumes a $\blacklozenge$-token. LiLi also stratify delaying actions into multiple levels. The delay caused by high-level actions may lead to executions of more low-level ones (and non-delaying actions), but *not* vice versa. Therefore we use w to represent the number of $\lozenge$-tokens, and use the vector u for the numbers of the k-level $\blacklozenge$-tokens, as defined in Fig. 9. They are described by the assertions $\lozenge(E)$ and $\blacklozenge(E_k,\ldots,E_1)$ respectively.

Rely/guarantee conditions and definite actions. The rely/guarantee conditions R and G specify stratified transitions between relational states, i.e., they are assertions over $(\mathfrak{S}, \mathfrak{S}', k)$, where k is the level of stratified delaying actions. $P \ltimes_k Q$ specifies a level-k delaying transition from P to Q. The subscript k can be omitted when $k = 0$. $[P]$ specifies identity transitions with the initial states satisfying P. $\lfloor G \rfloor_0$ can be satisfied only by level-0 transitions in G. Definite actions $\mathcal{D}$ is a special form of rely/guarantee condition. $P \leadsto Q$ specifies a transition where the final state satisfies Q if the initial state satisfies P.

As in LiLi, all assertions are implicitly parameterized with a thread ID t.

7.2 Logic Rules

Figure 10 shows the key logic rules. As the top rule of the logic, the OBJ rule says that, to verify Π satisfies its specification Γ with the object invariant P, one needs to specify the rely/guarantee conditions R and G, and the definite actions $\mathcal{D}$, and then prove that each individual object method implementation refines its specification. Here Γ must be an *atomic partial specification.*

For each method, we take the object invariant P and the annotated preconditions $\mathcal{P}$ and $\mathcal{P}'$ (in Π and Γ) as preconditions. We also assign $\blacklozenge$-tokens in the precondition ($\blacklozenge(E_k, \ldots, E_1)$) to constrain the number of delaying actions executed in the method. $\mathsf{arem}(C')$ says that the high-level code which remains to be refined at this point is C'. At the end we need to re-establish the object invariant P, and show that there is no more high-level code that needs to be refined (i.e., $\mathsf{arem}(\mathbf{skip})$), which means the method body indeed refines the specification C'.

The object invariant P should also ensure that the annotated pre-conditions $\mathcal{P}$ and $\mathcal{P}'$ are either both true or both false. That is, whenever P holds, it is either safe to call the methods at both the concrete and abstract levels, or unsafe to do so at both levels. The other side conditions in the rule are the same as those in LiLi and irrelevant to our extensions, so we omit the explanations here.

The WHL *rule.* The rule for **while**-loops is almost the same as the WHL rule in LiLi, with the changes highlighted in gray boxes. We verify the loop body with a precondition p', which needs to be derived from the loop invariant p and the loop condition B. In two cases we must ensure that there are no infinite loops:

- the definite action $\mathcal{D}$ is enabled (see Fig. 9 for the definition of $\mathsf{Enabled}(\mathcal{D})$). Then the loop must terminate to guarantee that the definite action $\mathcal{D}$ definitely occurs.
- the current thread is not blocked. Here we need to find a condition Q that ensures the current thread can make progress without waiting for actions of other threads.

The second premise of the rule says, in either of the two cases above we must consume a $\lozenge$-token for each round of the loop, as p' has one less token than $p \wedge B$.

On the other hand, if the current thread is blocked (Q does not hold) and it is not in the middle of a definite action, the loop can run an indefinite number of rounds to wait for the environment actions. It does not have to consume tokens. However, we must ensure the thread cannot be blocked forever, i.e., Q cannot be always false. This is achieved by the *definite-progress* condition introduced in LiLi. We show a generalized definition in Def. 7.1, with the changes highlighted in gray boxes.

Definition 7.1 (Definite Progress). $\mathfrak{S} \models (R, G\colon \mathcal{D} \xrightarrow{f} (Q, B_h))$ iff the following hold for any t:

(1) either $\mathfrak{S} \models Q_{\mathrm{t}}$, or $\mathfrak{S} \models \neg B_h$, or there exists t$'$ such that t$' \neq$ t and $\mathfrak{S} \models \mathsf{Enabled}(\mathcal{D}_{\mathrm{t}'})$;

(2) for any t$' \neq$ t and $\mathfrak{S}'$, if $(\mathfrak{S}, \mathfrak{S}', 0) \models R_{\mathrm{t}} \wedge (\langle \mathcal{D}_{\mathrm{t}'} \rangle \vee ((\neg B_h) \ltimes B_h))$, then $f_{\mathrm{t}}(\mathfrak{S}') < f_{\mathrm{t}}(\mathfrak{S})$;

(3) for any $\mathfrak{S}'$, if $(\mathfrak{S}, \mathfrak{S}', 0) \models R_{\mathrm{t}} \vee G_{\mathrm{t}}$, then $f_{\mathrm{t}}(\mathfrak{S}') \leq f_{\mathrm{t}}(\mathfrak{S})$.

Here f is a function that maps the relational states $\mathfrak{S}$ to some metrics over which there is a well-founded order $<$.

The definite progress condition $(R, G\colon \mathcal{D} \xrightarrow{f} (Q, B_h))$ tries to ensure Q is eventually always true, unless B_h is eventually always false. It requires the following conditions to hold:

(1) Either Q holds, which means that the low-level code is no longer blocked; or the high-level specification $\mathbf{await}(B_h)\{C'\}$ is disabled, so that the low-level code does not have to progress to refine the high-level code; or one of the definite actions in $\mathcal{D}$ that the current thread t

$$\frac{\begin{array}{c}\text{for all } f \in \mathit{dom}(\Pi): \quad \Pi(f) = (\mathcal{P}, x, C) \quad \Gamma(f) = (\mathcal{P}', y, C') \quad P \Rightarrow (\mathcal{P} \wedge \mathcal{P}') \vee (\neg\mathcal{P} \wedge \neg\mathcal{P}') \\ \mathcal{D}, R, G \vdash \{P \wedge (\mathcal{P} \wedge \mathcal{P}') \wedge (x = y) \wedge \mathsf{arem}(C') \wedge \blacklozenge(E_k, \ldots, E_1)\}\, C\, \{P \wedge \mathsf{arem}(\mathbf{skip})\} \\ \forall \mathrm{t}, \mathrm{t}'.\ \mathrm{t} \neq \mathrm{t}' \Longrightarrow G_{\mathrm{t}} \Rightarrow R_{\mathrm{t}'} \quad \mathsf{wffAct}(R, \mathcal{D}) \quad P \Rightarrow \neg\mathsf{Enabled}(\mathcal{D})\end{array}}{\mathcal{D}, R, G \vdash \{P\}\Pi : \Gamma} \ (\textsc{obj})$$

$$\frac{\begin{array}{c} p \wedge B \Rightarrow p' \quad p \wedge B \wedge (\mathsf{Enabled}(\mathcal{D}) \vee Q) \Rightarrow p' * (\lozenge \wedge \mathsf{emp}) \quad \mathcal{D}, R, G \vdash \{p'\}C\{p\} \\ p \wedge B \Rightarrow J \wedge \mathsf{arem}(\mathbf{await}(B')\{C'\}) \quad \mathsf{Sta}(J, R \vee G) \quad J \Rightarrow (R, G \colon \mathcal{D}' \xrightarrow{f} (Q, B')) \\ \mathcal{D}' \leqslant \mathcal{D} \quad \mathsf{wffAct}(R, \mathcal{D}')\end{array}}{\mathcal{D}, R, G \vdash \{p\}\mathbf{while}\ (B)\{C\}\{p \wedge \neg B\}} \ (\textsc{whl})$$

$$\frac{\begin{array}{c} p \wedge \mathsf{Enabled}(\mathcal{D}) \Rightarrow B \quad \mathcal{D}, \mathsf{Id}, G \vdash \{p \wedge B\}\langle C\rangle\{q\} \quad \mathsf{Sta}(\{p, q\}, R) \\ \mathcal{D}' \leqslant \mathcal{D} \quad \mathsf{wffAct}(R, \mathcal{D}') \quad p \Rightarrow \exists B', C'.\, \mathsf{arem}(\mathbf{await}(B')\{C'\}) \wedge (R \colon \mathcal{D}' \overset{f}{\circ\!\!\longrightarrow} (B, B'))\end{array}}{\mathcal{D}, R, G \vdash_{\mathsf{wfair}} \{p\}\mathbf{await}(B)\{C\}\{q\}} \ (\textsc{await-w})$$

$$\frac{\begin{array}{c} p \wedge \mathsf{Enabled}(\mathcal{D}) \Rightarrow B \quad \mathcal{D}, \mathsf{Id}, G \vdash \{p \wedge B\}\langle C\rangle\{q\} \quad \mathsf{Sta}(\{p, q\}, R) \\ \mathcal{D}' \leqslant \mathcal{D} \quad \mathsf{wffAct}(R, \mathcal{D}') \quad p \Rightarrow \exists B', C'.\, \mathsf{arem}(\mathbf{await}(B')\{C'\}) \wedge (R \colon \mathcal{D}' \overset{f}{\bullet\!\!\longrightarrow} (B, B'))\end{array}}{\mathcal{D}, R, G \vdash_{\mathsf{sfair}} \{p\}\mathbf{await}(B)\{C\}\{q\}} \ (\textsc{await-s})$$

Fig. 10. The key extensions of inference rules.

waits for is enabled in some thread t′. Here $\mathcal{D}$ can be viewed as a set of n definite actions in the form of $\mathcal{D}_1 \wedge \ldots \wedge \mathcal{D}_n$ parameterized with thread IDs.

(2) There is a well-founded metric f that becomes strictly smaller whenever (a) an environment thread t′ executes a definite action in $\mathcal{D}$, or (b) an environment action has turned the high-level command from disabled to enabled. Case (a) requires that the number of definite actions waited by the current thread must be strictly decreasing. Therefore eventually there are no enabled definite actions. By condition (1) we know eventually either Q or $\neg B_h$ is true. Case (b) further requires that the high-level command cannot be infinitely often disabled and then enabled during the loop. Therefore either B_h is eventually always true or it is eventually always false. In the former case we know Q must be eventually always true by the above condition (1). In the latter the loop does not have to terminate because the execution is well-blocked (see Fig. 6).

(3) The value of f over program states cannot be increased by any level-0 actions (i.e., non-delaying actions).

Note that the last two conditions do not prevent delaying actions (level-k actions where $k > 0$) from increasing the value of f, but such an increase can only occur a finite number of times because each delaying action consumes a $\blacklozenge$-token. The effects of delaying actions are shown in the ATOM rule, which is the same as in LiLi. Since the way delaying action is handled is orthogonal to our extension for partial methods, we omit the rule here.

To ensure the definite progress condition always holds, we need to find an invariant J which is preserved by any program step (by the current thread or by the environment), and require that J implies definite progress, given the currently remaining high-level command $\mathbf{await}(B')\{C'\}$. Note that to simplify the presentation we treat arem(**skip**) as arem(**await**(true){**skip**}) so that it can be reasoned about in the same way.

The WHL rule also allows us to use $\mathcal{D}'$, a subset of $\mathcal{D}$, to prove definite progress, which is useful to simplify the proofs. See the definition of $\mathcal{D}' \preccurlyeq \mathcal{D}$ in Fig. 9. $\mathcal{D}'$ also needs to satisfy $\mathsf{wffAct}(R, \mathcal{D}')$. This premise is taken from LiLi and we do not explain it here.

Since we have highlighted the changes over the WHL rule in LiLi, we can see that the WHL rule in LiLi is a specialization of ours when the high-level code is always in the form of **await**(true)$\{C'\}$.

Rules for **await** *commands.* Our logic introduces two new rules, AWAIT-W and AWAIT-S, to verify **await** commands in the *object implementation* under weakly fair and strongly fair scheduling. We use the subscripts of the judgment to distinguish the scheduling.

Naturally the AWAIT-W rule combines the ATOM rule in LiLi and the WHL rule. If **await**$(B)\{C\}$ is enabled, we can simply treat C as an atomic block $\langle C\rangle$ and apply the ATOM rule of LiLi to verify it. In this case we do not need to consider the interference and take Id as the rely condition (where Id is a shorthand notation for [true], which specifies arbitrary identity transitions).

Similar to the WHL rule, if the definite action $\mathcal{D}$ is enabled, then **await**$(B)\{C\}$ must be enabled at this point (see the first premise of the AWAIT-W rule). This is because we require that, when enabled, the definite action $\mathcal{D}$ must be fulfilled regardless of environment behaviors. Therefore the current thread cannot be blocked.

Finally, we require that, even if the command is blocked, it must be eventually enabled unless the corresponding high-level specification is blocked too. So if we view the enabling condition B the same as the condition Q we use in the WHL rule, we require the same *definite progress* condition, except that things are simpler here because **await**$(B)\{C\}$ finishes in one step once enabled, unlike loops which take multiple steps to finish even if Q holds. Therefore we do not need the invariant J used in the WHL rule, and we do not need to consider actions in G in the definite progress condition. We can use a simpler condition $(R\colon \mathcal{D}' \overset{f}{\circ\!\!\longrightarrow} (B, B'))$ defined below, which simply instantiates G with Id and Q with B in $(R, G\colon \mathcal{D}' \xrightarrow{f} (Q, B'))$ (see Def. 7.1).

Definition 7.2 (Definite Progress for Await).

- $\mathfrak{S} \models (R\colon \mathcal{D} \overset{f}{\circ\!\!\longrightarrow} (B_l, B_h))$ iff $\mathfrak{S} \models (R, \mathsf{Id}\colon \mathcal{D} \xrightarrow{f} (B_l, B_h))$.
- $\mathfrak{S} \models (R\colon \mathcal{D} \overset{f}{\bullet\!\!\longrightarrow} (B_l, B_h))$ iff the following hold for any t:
  1. either $\mathfrak{S} \models B_l$, or $\mathfrak{S} \models \neg B_h$, or there exists t$'$ such that t$' \neq$ t and $\mathfrak{S} \models \mathsf{Enabled}(\mathcal{D}_{t'})$;
  2. for any t$' \neq$ t and $\mathfrak{S}'$, if $(\mathfrak{S}, \mathfrak{S}', 0) \models R_t \land (\langle \mathcal{D}_{t'} \rangle \lor ((\neg B_h) \ltimes B_h))$, then $f_t(\mathfrak{S}') < f_t(\mathfrak{S})$;
  3. for any $\mathfrak{S}'$, if $(\mathfrak{S}, \mathfrak{S}', 0) \models R_t \land ((\neg B_l) \ltimes (\neg B_l))$, then $f_t(\mathfrak{S}') \leq f_t(\mathfrak{S})$.

The AWAIT-S rule for strongly fair scheduling looks almost the same as the AWAIT-W rule, with a slightly different definite progress condition $(R\colon \mathcal{D}' \overset{f}{\bullet\!\!\longrightarrow} (B, B'))$, which is also shown in Def. 7.2 with the difference highlighted in the gray box. The key difference here is that the low-level enabling condition B (represented as B_l in Def. 7.2) does not have to be stable once it becomes true. Under strongly fair scheduling we know the **await** block will be executed as long as it is enabled infinitely often. Therefore in condition (3) we only need to ensure that f does not increase if the enabling condition B_l remains false, but we allow f to increase whenever we see B_l holds.

The WHL rule, AWAIT-W rule and AWAIT-S rule are the only new command rules we introduce to reason about partial methods and blocking primitives. All the other command rules are taken directly from LiLi, which are omitted here.

7.3 Soundness of the Logic

The two AWAIT rules actually give us two program logics, for strongly fair and weakly fair scheduling respectively. To distinguish them, we use $\mathcal{D}, R, G \vdash_{\chi} \{P\}\Pi : \Gamma$ to represent the verification using the logic for χ-scheduling ($\chi \in \{\mathsf{sfair}, \mathsf{wfair}\}$), where the corresponding AWAIT rule is used.

Theorem 7.3 shows that our logic is sound in that it guarantees linearizability and partial deadlock freedom (PDF) of concurrent objects. It also ensures partial starvation freedom (PSF) if the rely/guarantee conditions specify only level-0 actions, as required by $R \Rightarrow \lfloor R \rfloor_0$ and $G \Rightarrow \lfloor G \rfloor_0$. That is, none of the object actions of a thread could delay the progress of other threads. With the specialized R and G, we can derive the progress of each single thread, which gives us PSF.

Theorem 7.3 (Soundness). *If* $\mathcal{D}, R, G \vdash_{\chi} \{P\}\Pi : \Gamma$ *and* $\varphi \Rightarrow P$, *then*

(1) both $\Pi \preceq^{\mathsf{lin}}_{\varphi} \Gamma$ *and* $\mathsf{PDF}^{\chi}_{\varphi,\Gamma}(\Pi)$ *hold; and*

(2) if $R \Rightarrow \lfloor R \rfloor_0$ *and* $G \Rightarrow \lfloor G \rfloor_0$, *then* $\mathsf{PSF}^{\chi}_{\varphi,\Gamma}(\Pi)$ *holds.*

where $\chi \in \{\mathsf{sfair}, \mathsf{wfair}\}$, *and* $\varphi \Rightarrow P \overset{\text{def}}{=} \forall \sigma, \Sigma.\, (\varphi(\sigma) = \Sigma) \Longrightarrow (\sigma, \Sigma) \models P$.

Proofs of the theorem are in the TR [Liang and Feng 2017]. We first prove the logic establishes the progress-aware contextual refinements, and then apply the Abstraction Theorem 6.2 to ensure linearizability and the progress properties. The proof structure is similar to the one for LiLi.

8 EXAMPLES

We have applied the program logic to verify ticket locks [Mellor-Crummey and Scott 1991], test-and-set locks [Herlihy and Shavit 2008], bounded partial queues with two locks [Herlihy and Shavit 2008] (where the locks are implemented using the specification (2.3)) and Treiber stacks [Treiber 1986] with partial pop methods. Perhaps interestingly, we also use our logic to prove that, for the atomic partial specification Γ for locks, the wrapping of Γ (as the object implementation) respects Γ itself as the atomic specification under the designated fairness conditions, i.e., $\left(\mathcal{D}, R, G \vdash \{P\}\mathsf{wr}^{\chi}_{\mathsf{Prog}}(\Gamma) : \Gamma\right)$ holds for certain $\mathcal{D}$, R, G and P, and for different combinations of fairness χ and progress Prog. This result validates our wrappers and program logic. It shows $\mathsf{Prog}^{\chi}_{\varphi,\Gamma}(\mathsf{wr}^{\chi}_{\mathsf{Prog}}(\Gamma))$ holds, i.e., each wrapper itself satisfies the corresponding progress property. Below we show the proofs for test-and-set locks, ticket locks, and simple locks implemented using **await** which guarantee PSF under weak fairness. The proofs of other examples are given in our TR.

8.1 Test-and-Set Locks

In Fig. 11, we verify PDF of the test-and-set locks using our logic with the atomic partial specifications `L_ACQ'` and `L_REL` defined in (2.3). To distinguish the variables at the two levels, below we use capital letters (e.g., `L`) in the specifications and small letters (e.g., `l`) in the implementations.

As we explained in Sec. 3, the method `L_rel` and the specification `L_REL` have annotated preconditions (`l = cid`) and (`L = cid`), respectively. That is, it is not allowed to call `L_rel` (or `L_REL`) when the thread does not hold the lock. The annotated precondition for `L_acq` and `L_ACQ'` is true. In Fig. 11, we define the assertion lock as the object invariant P used in the OBJ rule. Then the method `L_acq` is verified with the precondition lock, and `L_rel` is verified with the precondition lock $\wedge$ (`l = cid`) which is reduced to $\mathsf{lockedBy}_{\mathtt{cid}}$, as shown in Fig. 11.

To verify `L_acq`, we make the following key observations. When the **cas** at line 3 succeeds, `L_ACQ'` must be enabled and can be executed correspondingly. And at the time when the **cas** fails, `L_ACQ'` must be disabled. The progress of `L_acq` relies on that the environment thread holding the lock could eventually release the lock, i.e., turning the current thread's `L_ACQ'` from disabled to enabled. But such an action is not "definite", since the client thread may never call the `L_rel`

$$\mathsf{lock} \overset{\text{def}}{=} \exists s.\ \mathsf{lock}_s \qquad \mathsf{lock}_s \overset{\text{def}}{=} (\mathtt{l} = \mathtt{L} = s)$$
$$\mathsf{unlocked} \overset{\text{def}}{=} \mathsf{lock}_0$$
$$\mathsf{locked} \overset{\text{def}}{=} \exists \mathrm{t}.\ \mathsf{lockedBy}_{\mathrm{t}}$$
$$\mathsf{lockedBy}_{\mathrm{t}} \overset{\text{def}}{=} \mathsf{lock}_{\mathrm{t}} \wedge (\mathrm{t} \neq 0)$$
$$R_{\mathrm{t}} \overset{\text{def}}{=} \bigvee_{\mathrm{t}' \neq \mathrm{t}} G_{\mathrm{t}'} \qquad G_{\mathrm{t}} \overset{\text{def}}{=} Acq_{\mathrm{t}} \vee Rel_{\mathrm{t}}$$
$$Acq_{\mathrm{t}} \overset{\text{def}}{=} \mathsf{unlocked} \ltimes_1 \mathsf{lockedBy}_{\mathrm{t}}$$
$$Rel_{\mathrm{t}} \overset{\text{def}}{=} \mathsf{lockedBy}_{\mathrm{t}} \ltimes_0 \mathsf{unlocked}$$
$$\mathcal{D} \overset{\text{def}}{=} \mathsf{false} \rightsquigarrow \mathsf{true}$$
$$J \overset{\text{def}}{=} \mathsf{lock}$$
$$Q \overset{\text{def}}{=} \mathsf{unlocked}$$
$$f(\mathfrak{S}) = \begin{cases} 1 & \text{if } \mathfrak{S} \models \mathsf{locked} \\ 0 & \text{if } \mathfrak{S} \models Q \end{cases}$$

```
L_acq(){
    {lock ∧ ♦ ∧ arem(L_ACQ')}
1   local b := false;
    {((¬b) ∧ lock ∧ ♦ ∧ ◊ ∧ arem(L_ACQ'))
         ∨ (b ∧ lockedBy_cid ∧ arem(skip))}
2   while (!b) {
      {((unlocked ∧ ♦) ∨ (locked ∧ ♦ ∧ ◊))
       ∧ arem(L_ACQ')}
3     b := cas(&l, 0, cid);
4   }
    {lockedBy_cid ∧ arem(skip)}
}
L_rel(){
    {lockedBy_cid ∧ arem(L_REL)}
5   l := 0;
    {lock ∧ arem(skip)}
}
```

Fig. 11. Proofs for the test-and-set lock.

method. The definite action $\mathcal{D}$ for this object can be defined as false $\rightsquigarrow$ true, saying that there is no definite action that a thread needs to complete.

The action Acq_{t} (corresponding to the successful **cas** at line 3) is a delaying action (defined with level 1). When thread t succeeds in **cas**, termination of other threads' `L_acq` can be delayed, as allowed by PDF. The thread t has to pay a ♦-token, given in the precondition of `L_acq`.

The definite progress condition $(R, G\colon \mathcal{D} \xrightarrow{f} (Q, \mathtt{L=0}))$ now says that thread t is either at a state that it itself can progress (i.e., Q holds), or blocked at the abstract level (i.e., `L=0` does not hold). The metric $f_{\mathrm{t}}(\mathfrak{S})$ decreases when an environment thread releases `L`, but can be reset (which means thread t is delayed) if an environment thread successfully acquires the lock.

By the Soundness Theorem 7.3, we know the test-and-set lock object satisfies the PDF property, and contextually refines the abstraction generated by the corresponding PDF wrappers in Fig. 7, under strongly and weakly fair scheduling.

8.2 Ticket Locks

In Fig. 12, we prove the ticket lock object satisfies PSF. We introduce some write-only auxiliary variables to help the verification. First, we introduce an array `ticket` to help specify the queue of the threads requesting the lock. Each array cell `ticket[i]` records the ID of the unique thread getting the ticket number `i` (see line 2). Second, we introduce a lock bit `l` to make the lock acquirement and lock release explicit (see lines 4 and 5).

We then define the object invariant $\mathsf{lock}(s, tl, n_1, n_2)$. It says that the lock bits `l` and `L` are equal, n_1 and n_2 are the values of `owner` and `next` respectively, and tl is the list of the threads recorded in $\mathtt{ticket}[n_1]$, $\mathtt{ticket}[n_1+1]$, …, $\mathtt{ticket}[n_2-1]$ (as specified by $\mathsf{tickets}(tl, n_1, n_2)$).

The guarantee condition G_{t} describes the possible atomic actions of thread t. Req_{t} adds t at the end of tl of the threads requesting the lock and also increments `next`. It corresponds to line 2 in the code at the top of Fig. 12. Acq_{t} sets the lock bits to t, explicitly indicating the lock acquirement (see line 4). It is also a definite action (see the definition of $\mathcal{D}_{\mathrm{t}}$) since thread t must acquire the lock if its loop at line 3 terminates. Rel_{t} increments `owner` to dequeue the thread t which currently holds the lock, and resets the lock bits (see line 5). All actions are at level 0. There are no delaying actions.

```
tkL_acq(){                                                    tkL_rel(){
1  local i, o;                                                5  <owner := owner+1; l := 0 >;
2  <i := getAndInc(&next); ticket[i] := cid >;                }
3  o := owner;  while (i != o) { o := owner; }
4  l := cid;
}
```

$\mathsf{lock}(s, tl, n_1, n_2) \overset{\text{def}}{=}$

$\quad (\mathtt{l} = \mathsf{L} = s \wedge (s = \mathsf{head}(tl) \vee s = 0)) * ((\mathsf{owner} = n_1) * (\mathsf{next} = n_2) \wedge (n_1 \le n_2)) * \mathsf{tickets}(tl, n_1, n_2)$

$G_{\mathsf{t}} \overset{\text{def}}{=} Req_{\mathsf{t}} \vee Acq_{\mathsf{t}} \vee Rel_{\mathsf{t}} \qquad \mathcal{D}_{\mathsf{t}} \overset{\text{def}}{=} \forall tl, n_1, n_2.\ \mathsf{lock}(0, \mathsf{t} :: tl, n_1, n_2) \leadsto \mathsf{lock}(\mathsf{t}, \mathsf{t} :: tl, n_1, n_2)$

$Req_{\mathsf{t}} \overset{\text{def}}{=} \exists s, tl, n_1, n_2.\ \mathsf{lock}(s, tl, n_1, n_2) \ltimes \mathsf{lock}(s, tl{+\!+}[\mathsf{t}], n_1, n_2 + 1)$

$Acq_{\mathsf{t}} \overset{\text{def}}{=} \exists tl, n_1, n_2.\ \mathsf{lock}(0, \mathsf{t} :: tl, n_1, n_2) \ltimes \mathsf{lock}(\mathsf{t}, \mathsf{t} :: tl, n_1, n_2)$

$Rel_{\mathsf{t}} \overset{\text{def}}{=} \exists tl, n_1, n_2.\ \mathsf{lock}(\mathsf{t}, \mathsf{t} :: tl, n_1, n_2) \ltimes \mathsf{lock}(0, tl, n_1 + 1, n_2)$

$J_{\mathsf{t}} \overset{\text{def}}{=} \exists s, n_1, n_2, tl_1, tl_2.\ \mathsf{tlocked}_{tl_1, \mathsf{t}, tl_2}(s, n_1, \mathtt{i}, n_2) \wedge (\mathtt{o} \le n_1)$

$Q_{\mathsf{t}} \overset{\text{def}}{=} \exists n_2, tl_2.\ \mathsf{lock}(0, \mathsf{t} :: tl_2, \mathtt{i}, n_2) \wedge (\mathtt{o} \le \mathtt{i}) \qquad f(\mathfrak{S}) = \begin{cases} 2k+1 & \text{if } \mathfrak{S} \models (\mathtt{i} - \mathsf{owner} = k) * (\mathtt{l} = 0) \\ 2k & \text{if } \mathfrak{S} \models (\mathtt{i} - \mathsf{owner} = k) * (\mathtt{l} \neq 0) \end{cases}$

Fig. 12. Proofs for the ticket lock (with auxiliary code in gray).

By applying the WHL rule of our logic, we need to prove the definite progress condition $J \Rightarrow (R, \mathsf{Id} \colon \mathcal{D} \xrightarrow{f} (Q, \mathtt{L{=}0}))$ for the loop at line 3. Here J, Q and f are defined at the bottom of Fig. 12. In the definition of J_{t}, we use $\mathsf{tlocked}_{tl_1, \mathsf{t}, tl_2}(s, n_1, \mathtt{i}, n_2)$ to say that t is requesting the lock and its ticket number is i. Here tl_1 is the list of the threads which are waiting ahead of t, and tl_2 is for the threads behind t. Q_{t} specifies the case when tl_1 is empty. In this case the lock bits must be 0 and tlocked is reduced to lock, as shown at the bottom of Fig. 12.

The metric $f_{\mathsf{t}}(\mathfrak{S})$ is determined by the number of threads ahead of t in the waiting queue and the status of the lock bits. It decreases when an environment thread t′ does the definite action $\mathcal{D}_{\mathsf{t}'}$, setting the lock bits to t′. It also decreases when t′ releases the lock and increments `owner`, turning $(\mathsf{L} \neq 0)$ to $(\mathsf{L} = 0)$. Thus we can prove $J \Rightarrow (R, \mathsf{Id} \colon \mathcal{D} \xrightarrow{f} (Q, \mathtt{L{=}0}))$.

By the Soundness Theorem 7.3, we know the ticket lock object satisfies the PSF property, and contextually refines the abstraction generated by the corresponding PSF wrappers, under both strongly and weakly fair scheduling. The detailed formal proofs are given in the TR.

8.3 Simple PSF Locks with Await Blocks

Figure 13 shows the proofs of a simple lock object implemented with an **await** statement which guarantees PSF under weak fairness. The `acquire` method is simply $\mathsf{wr}^{\mathsf{wfair}}_{\mathsf{PSF}}$(**await**(`l=0`){`l:=cid`}). The `release` method resets the lock bit `l` directly. It has the annotated precondition (`l` = `cid`). We still verify the object in our logic with the specifications `L_ACQ'` and `L_REL` defined in (2.3).

We first define the object invariant P used in the OBJ rule. It is defined based on lock, which requires `l` to have the same value as the abstract lock L. The queue `listid` records the threads currently waiting for the lock. Here $\mathsf{diff}(tb)$ says that the threads in tb are all different. Then the object invariant P_{t} further requires that the current thread t is not recorded in `listid`. It is preserved before and after t calls a method.

The object has three kinds of possible actions (see the definition of G). Req_{t} appends the thread t at the end of `listid` to request the lock (line 1). Acq_{t} acquires the lock if the lock is available and

$$P_{\mathrm{t}} \overset{\text{def}}{=} \exists s, tb.\ \mathsf{lock}_s(tb) \wedge (\mathrm{t} \notin tb) \qquad \text{where } tb ::= \epsilon \mid (\mathrm{t}, \text{`}\mathtt{l=0}\text{'}) :: tb$$

$$\mathsf{lock}_s(tb) \overset{\text{def}}{=} (\mathtt{l} = \mathsf{L} = s) * (\mathtt{listid} = tb) \wedge \mathsf{diff}(tb)$$

$$\mathsf{unlocked}(tb) \overset{\text{def}}{=} \mathsf{lock}_0(tb) \qquad \mathsf{lockReq}_{\mathrm{t}} \overset{\text{def}}{=} \exists s, tb.\ \mathsf{lock}_s(tb) \wedge (\mathrm{t} \in tb)$$

$$\mathsf{locked}_{\mathrm{t}}(tb) \overset{\text{def}}{=} \mathsf{lock}_{\mathrm{t}}(tb) \wedge (\mathrm{t} \neq 0) \wedge (\mathrm{t} \notin tb) \qquad \mathsf{locked}_{\mathrm{t}} \overset{\text{def}}{=} \exists tb.\ \mathsf{locked}_{\mathrm{t}}(tb)$$

$$G_{\mathrm{t}} \overset{\text{def}}{=} Req_{\mathrm{t}} \vee Acq_{\mathrm{t}} \vee Rel_{\mathrm{t}} \qquad \mathcal{D}_{\mathrm{t}} \overset{\text{def}}{=} \forall tb.\ \mathsf{unlocked}((\mathrm{t}, \text{`}\mathtt{l=0}\text{'}) :: tb) \rightsquigarrow \mathsf{locked}_{\mathrm{t}}(tb)$$

$$Req_{\mathrm{t}} \overset{\text{def}}{=} \exists s, tb.\ (\mathsf{lock}_s(tb) \wedge (\mathrm{t} \notin tb)) \ltimes \mathsf{lock}_s(tb +\!\!+ [(\mathrm{t}, \text{`}\mathtt{l=0}\text{'})])$$

$$Acq_{\mathrm{t}} \overset{\text{def}}{=} \exists tb.\ \mathsf{unlocked}((\mathrm{t}, \text{`}\mathtt{l=0}\text{'}) :: tb) \ltimes \mathsf{locked}_{\mathrm{t}}(tb) \qquad Rel_{\mathrm{t}} \overset{\text{def}}{=} \exists tb.\ \mathsf{locked}_{\mathrm{t}}(tb) \ltimes \mathsf{unlocked}(tb)$$

$$f_{\mathrm{t}}(\mathfrak{S}) \overset{\text{def}}{=} \begin{cases} 2k+1 & \text{if } \exists s, tb, tb'.\ (\mathfrak{S} \models \mathsf{lock}_s(tb +\!\!+ [(\mathrm{t}, \text{`}\mathtt{l=0}\text{'})] +\!\!+ tb') \wedge s \neq 0) \wedge |tb| = k \\ 2k & \text{if } \exists tb, tb'.\ (\mathfrak{S} \models \mathsf{unlocked}(tb +\!\!+ [(\mathrm{t}, \text{`}\mathtt{l=0}\text{'})] +\!\!+ tb')) \wedge |tb| = k \end{cases}$$

```
acquire(){
    {$P_{\mathtt{cid}} \wedge \mathsf{arem}(\mathsf{L\_ACQ'})$}
  1 listid := listid ++ [(cid, 'l=0')];
    {$\mathsf{lockReq}_{\mathtt{cid}} \wedge \mathsf{arem}(\mathsf{L\_ACQ'})$}
  2 await (l = 0 /\ cid = enhd(listid)) {
      {$\exists tb.\ \mathsf{unlocked}((\mathtt{cid}, \text{`}\mathtt{l=0}\text{'}) :: tb) \wedge \mathsf{arem}(\mathsf{L\_ACQ'})$}
  3   l := cid;  listid := listid \ cid;
      {$\exists tb.\ \mathsf{locked}_{\mathtt{cid}}(tb) \wedge \mathsf{arem}(\mathbf{skip})$}
  4 }
    {$\mathsf{locked}_{\mathtt{cid}} \wedge \mathsf{arem}(\mathbf{skip})$}
}
```

```
release(){
    {$\mathsf{locked}_{\mathtt{cid}} \wedge \mathsf{arem}(\mathsf{L\_REL})$}
  5 l := 0;
    {$P_{\mathtt{cid}} \wedge \mathsf{arem}(\mathbf{skip})$}
}
```

Fig. 13. Proofs for the simple PSF lock under weak fairness.

t is at the head of `listid` (lines 2-4). Rel_{t} releases the lock (line 5). Here Acq_{t} is also the definite action of thread t (see the definition of $\mathcal{D}$). None of the actions are delaying actions.

To verify the **await** statement at lines 2-4, we apply the AWAIT-W rule in Fig. 10, and prove:

$$\mathsf{lockReq} \Rightarrow (R\colon \mathcal{D} \overset{f}{\circ\!\!\longrightarrow} (\mathtt{l = 0} \wedge \mathtt{cid = enhd(listid)}, \mathsf{L} = 0)) . \tag{8.1}$$

The metric f is defined at the top of Fig. 13. We can see that $f_{\mathrm{t}}(\mathfrak{S})$ decreases when an environment thread t′ performs a definite action, since $\mathcal{D}_{\mathrm{t'}}$ will remove t′ that is waiting ahead of the thread t. Also $f_{\mathrm{t}}(\mathfrak{S})$ decreases when t′ releases the lock, turning $(\mathsf{L} \neq 0)$ to $(\mathsf{L} = 0)$. Thus (8.1) holds.

By the Soundness Theorem 7.3, we know this simple lock satisfies PSF under weak fairness.

9 RELATED WORK AND CONCLUSION

There has been much work on the relationships between linearizability, progress properties and contextual refinement (e.g., [Filipović et al. 2009; Gotsman and Yang 2011, 2012; Liang et al. 2013]), and on verifying progress properties or progress-aware refinement (e.g., [Boström and Müller 2015; da Rocha Pinto et al. 2016; Gotsman et al. 2009; Hoffmann et al. 2013; Jacobs et al. 2015; Tassarotti et al. 2017]). But none of them studies objects with partial methods as we do. On the other hand, our ideas might be general enough to be integrated with these verification methods to support blocking primitives and partial methods. For instance, Tassarotti et al. [2017] propose a higher-order logic based on Iris [Jung et al. 2015] for fair refinements. Our wrappers and reasoning method may be applied there to support higher-order refinement reasoning with blocking primitives. The logic by Boström and Müller [2015] ensures that no thread will be blocked forever. It supports special built-in blocking primitives for locking, message passing and thread join. Their obligation-based reasoning strategies may be applied to **await** blocks too, to verify that the client threads of **await** will not be permanently blocked.

In our previous work we propose the program logic LiLi [Liang and Feng 2016] to verify starvation-free and deadlock-free objects. This work is inspired by several ideas from LiLi:

- The soundness of LiLi ensures a progress-aware contextual refinement, which gives starvation-freedom or deadlock-freedom, if fed with different abstractions generated by specific code wrappers. Here we take a similar approach, and define new wrappers to generate abstractions for PSF and PDF objects.
- LiLi sorts progress properties in two dimensions called blocking and delay, and distinguish starvation-freedom and deadlock-freedom by whether delay is permitted. Here the difference between our PSF and PDF also lies in the delay dimension.
- The program logic proposed in this paper is a generalization of LiLi. Both logics use tokens to support delay, and use similar definite progress conditions to support blocking.

However, there are two main problems with LiLi, which are addressed in this paper:

- LiLi does not provide abstractions for objects with partial methods. When using LiLi to verify lock-based algorithms (such as the counters shown in Fig. 1(b) and (d) in this paper), one has to inline the implementations of locks, losing the modularity of verification. Here we define progress-aware abstractions for objects with partial methods, allowing us to verify their clients in a modular way.
- The inference rules of LiLi do not apply to objects with partial methods, such as the objects in Sec. 8 in this paper. We have explained the reasons and our solutions in Sec. 7.

Schellhorn et al. [2016] propose a proof method for verifying starvation-freedom. Their approach is based on a special predicate which describes the waiting-for relations among the threads. However, their work has similar problems as LiLi, and cannot apply to the examples considered in this paper.

Gu et al. [2016] verify progress of the ticket lock implementation as part of their verified kernel. Their specification of the lock relies on the behaviors of clients. It requires that the client owning a lock must eventually release it. Then they prove that the acquire method always terminates with the cooperative clients. It is unclear how the approach can be applied for general objects with partial methods.

Conclusions and more discussions. We have studied the progress of objects with partial methods in three aspects. First, we define new progress properties, partial starvation-freedom (PSF) and partial deadlock-freedom (PDF). Second, we design wrappers to generate abstractions for PSF and PDF objects under strongly or weakly fair scheduling. Third, we develop a program logic to verify PSF and PDF.

Although our program logic verifies both linearizability and progress properties, it is focused more on the latter. Existing work [Khyzha et al. 2017; Liang and Feng 2013; Turon et al. 2013] has shown that linearizability itself can be challenging to verify, and special mechanisms are needed for very fine-grained objects with non-fixed linearization points (LPs). Our logic cannot verify these objects, but our conjecture is that the mechanisms handling non-fixed LPs (as in [Liang and Feng 2013]) are orthogonal to our progress reasoning, and they can be integrated into our logic if needed.

The logic follows LiLi's ideas of definite actions and stratified tokens to reason about progress. They can be viewed as special strategies implementing the general principle for termination reasoning, that is to find a well-founded metric that keeps decreasing during the program execution. These ideas and rules give a concrete guide to users on how to construct the metric and the proofs. Although we have tried to make them as general as possible, and they have been shown applicable to many non-trivial algorithms [Liang and Feng 2016, 2017], they may not be complete and it would be unsurprising if there are examples that they cannot handle. As future work, we would like to verify more examples to explore the scope of the applicability.

The specifications of linearizable objects must be *atomic*, but sometimes we may want to give non-atomic specifications to object methods. We can apply our wrappers to every occurrence of the **await** blocks in the non-atomic specifications to establish progress-aware refinements. We suspect that our logic can still be used to verify such refinements (as in [Liang et al. 2014]). Another potential limitation may be due to the use of the pure Boolean expression B in $\textbf{await}(B)\{C\}$, which may limit the expressiveness of the specifications. However, our technical development does not rely on this setting. Everything may still hold if we replace B with the more expressive state assertions.

Other interesting future work includes automating the verification process. One of the key problems is to infer the definite actions and prove the definite progress conditions. There have been efforts to synthesize the ranking functions for loop termination (see [Cook et al. 2011] for an overview), which may provide insights for automating the definite progress proofs. In addition we might be able to follow the ideas in automated rely-guarantee reasoning (e.g., [Calcagno et al. 2007]) to automate the verification in our rely-guarantee logic.

ACKNOWLEDGMENTS

We would like to thank anonymous referees for their suggestions and comments on earlier versions of this paper. This work is supported in part by grants from National Natural Science Foundation of China (NSFC) under Grant Nos. 61379039, 61502442 and 61632005.

REFERENCES

Pontus Boström and Peter Müller. 2015. Modular Verification of Finite Blocking in Non-terminating Programs. In *Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP 2015)*. 639–663.

Cristiano Calcagno, Matthew J. Parkinson, and Viktor Vafeiadis. 2007. Modular Safety Checking for Fine-Grained Concurrency. In *Proceedings of the 14th International Symposium on Static Analysis (SAS 2007)*. 233–248.

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2011. Proving Program Termination. *Commun. ACM* 54, 5 (2011), 88–98.

Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. 2016. Modular Termination Verification for Non-blocking Concurrency. In *Proceedings of the 25th European Symposium on Programming Languages and Systems (ESOP 2016)*. 176–201.

Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. 2009. Abstraction for Concurrent Objects. In *Proceedings of the 18th European Symposium on Programming (ESOP 2009)*. 252–266.

Alexey Gotsman, Byron Cook, Matthew J. Parkinson, and Viktor Vafeiadis. 2009. Proving that Non-Blocking Algorithms Don't Block. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009)*. 16–28.

Alexey Gotsman and Hongseok Yang. 2011. Liveness-Preserving Atomicity Abstraction. In *Proceedings of the 38th International Conference on Automata, Languages and Programming (ICALP 2011)*. 453–465.

Alexey Gotsman and Hongseok Yang. 2012. Linearizability with Ownership Transfer. In *Proceedings of the 23rd International Conference on Concurrency Theory (CONCUR 2012)*. 256–271.

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI 2016)*. 653–669.

Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming.* Morgan Kaufmann.

Maurice Herlihy and Nir Shavit. 2011. On the Nature of Progress. In *Proceedings of the 15th International Conference on Principles of Distributed Systems (OPODIS 2011)*. 313–328.

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

Jan Hoffmann, Michael Marmar, and Zhong Shao. 2013. Quantitative Reasoning for Proving Lock-Freedom. In *Proceedings of the 28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2013)*. 124–133.

Bart Jacobs, Dragan Bosnacki, and Ruurd Kuiper. 2015. Modular Termination Verification. In *Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP 2015)*. 664–688.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*. 637–650.

Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew J. Parkinson. 2017. Proving Linearizability Using Partial Orders. In *Proceedings of the 26th European Symposium on Programming (ESOP 2017)*. 639–667.

Hongjin Liang and Xinyu Feng. 2013. Modular Verification of Linearizability with Non-Fixed Linearization Points. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*. 459–470.

Hongjin Liang and Xinyu Feng. 2016. A Program Logic for Concurrent Objects Under Fair Scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*. 385–399.

Hongjin Liang and Xinyu Feng. 2017. *Progress of Concurrent Objects with Partial Methods (Extended Version)*. Technical Report. https://cs.nju.edu.cn/hongjin/papers/popl18-partial-tr.pdf.

Hongjin Liang, Xinyu Feng, and Zhong Shao. 2014. Compositional Verification of Termination-preserving Refinement of Concurrent Programs. In *Proceedings of the Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science (CSL-LICS 2014)*. 65:1–65:10.

Hongjin Liang, Jan Hoffmann, Xinyu Feng, and Zhong Shao. 2013. Characterizing Progress Properties of Concurrent Objects via Contextual Refinements. In *Proceedings of the 24th International Conference on Concurrency Theory (CONCUR 2013)*. 227–241.

John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (1991), 21–65.

Gerhard Schellhorn, Oleg Travkin, and Heike Wehrheim. 2016. Towards a Thread-Local Proof Technique for Starvation Freedom. In *Proceedings of the 12th International Conference on Integrated Formal Methods (IFM 2016)*. 193–209.

Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *Proceedings of the 26th European Symposium on Programming (ESOP 2017)*. 909–936.

R. Kent Treiber. 1986. *System Programming: Coping with Parallelism.* Technical Report RJ 5118. IBM Almaden Research Center.

Aaron Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013. Logical Relations for Fine-Grained Concurrency. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013)*. 343–356.