A Program Logic for Concurrent Objects under Fair Scheduling

Hongjin Liang Xinyu Feng

School of Computer Science and Technology & Suzhou Institute for Advanced Study
University of Science and Technology of China
lhj1018@ustc.edu.cn xyfeng@ustc.edu.cn

Abstract

Existing work on verifying concurrent objects is mostly concerned with safety only, e.g., partial correctness or linearizability. Although there has been recent work verifying lock-freedom of non-blocking objects, much less efforts are focused on deadlock-freedom and starvation-freedom, progress properties of blocking objects. These properties are more challenging to verify than lock-freedom because they allow the progress of one thread to depend on the progress of another, assuming fair scheduling.

We propose LiLi, a new rely-guarantee style program logic for verifying linearizability and progress *together* for concurrent objects under fair scheduling. The rely-guarantee style logic unifies thread-modular reasoning about both starvation-freedom and deadlock-freedom in one framework. It also establishes progress-aware abstraction for concurrent objects, which can be applied when verifying safety and liveness of client code. We have successfully applied the logic to verify starvation-freedom or deadlock-freedom of representative algorithms such as ticket locks, queue locks, lock-coupling lists, optimistic lists and lazy lists.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification – Correctness proofs, Formal methods; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Theory, Verification

Keywords Concurrency, Progress, Program Logic, Rely-Guarantee Reasoning, Refinement

1. Introduction

A concurrent object or library provides a set of methods that allow multiple client threads to manipulate the shared data structure. Blocking synchronization (i.e., mutual exclusion locks), as a straightforward technique to ensure exclusive accesses and to control the interference, has been widely-used in object implementations to achieve linearizability, which ensures the object methods behave as atomic operations in a concurrent setting.

POPL'16, January 20–22, 2016, St. Petersburg, FL, USA
© 2016 ACM. 978-1-4503-3549-2/16/01...$15.00
http://dx.doi.org/10.1145/2837614.2837635

In addition to linearizability, a safety property, object implementations are expected to also satisfy progress properties. The non-blocking progress properties, such as wait-freedom and lock-freedom which have been studied a lot (e.g., [5, 10, 16, 24]), guarantee the termination of the method calls independently of how the threads are scheduled. Unfortunately these properties are too strong to be satisfied by algorithms with blocking synchronization. For clients using lock-based objects, a delay of a thread holding a lock will block other threads requesting the lock. Thus their progress relies on the assumption that every thread holding the lock will eventually be scheduled to release it. This assumption requires *fair* scheduling, i.e., every thread gets eventually executed. As summarized by Herlihy and Shavit [14], there are two desirable progress properties for blocking algorithms, both assuming fair scheduling:

- *Deadlock-freedom*: In each fair execution, there always exists some method call that can finish. It disallows the situation in which multiple threads requesting locks are waiting for each other to release the locks in hand. It ensures the absence of livelock, but not starvation.
- *Starvation-freedom*: Every method call should finish in fair executions. It requires that every thread attempting to acquire a lock should eventually succeed and in the end release the lock. Starvation-freedom is stronger than deadlock-freedom. Nevertheless it can often be achieved by using fair locks [13].

Recent program logics for verifying concurrent objects either prove only linearizability and ignore the issue of termination (e.g., [6, 21, 30, 31]), or aim for non-blocking progress properties (e.g., [5, 10, 16, 24]), which cannot be applied to blocking algorithms that progress only under fair scheduling. The fairness assumption introduces complicated interdependencies among progress properties of threads, making it incredibly more challenging to verify the lock-based algorithms than their non-blocking counterparts. We will explain the challenges in detail in Sec. 2.

It is important to note that, although there has been much work on deadlock detection or deadlock-freedom verification (e.g., [4, 20, 32]), deadlock-freedom is often defined as a safety property, which ensures the lack of circular waiting for locks. It does not prevent livelock or non-termination inside the critical section. Another limitation of this kind of work is that it often assumes built-in lock primitives, and lacks support of ad-hoc synchronization (e.g., mutual exclusion achieved using spin-locks implemented by the programmers). The deadlock-freedom we discuss in this paper is a liveness property and its definition does not rely on built-in lock primitives. We discuss more related work on liveness verification in Sec. 8.

In this paper we propose LiLi, a new rely-guarantee style logic for concurrent objects under fair scheduling. LiLi is the first program logic that unifies verification of linearizability, starvation-freedom and deadlock-freedom in one framework (the name LiLi stands for

Linearizability and Liveness). It supports verification of both mutex-based pessimistic algorithms (including fine-grained ones such as lock-coupling lists) and optimistic ones such as optimistic lists and lazy lists. The unified approach allows us to prove *in the same logic*, for instance, the lock-coupling list algorithm is starvation-free if we use fair locks, e.g., ticket locks [25], and is deadlock-free if regular test-and-set (TAS) based spin locks are used. Our work is based on earlier work on concurrency verification, but we make the following new contributions:

- We divide environment interference that affects progress of a thread into two classes, namely *blocking* and *delay*. We show different occurrences of them correspond to the classification of progress into wait-freedom, lock-freedom, starvation-freedom and deadlock-freedom (see Sec. 2.2.1 and Sec. 6). Recognizing the two classes of interference allows us to come up with different mechanisms in our program logic to reason about them separately. Our logic also provides parameterized specifications, which can be instantiated to choose different combinations of the mechanisms. This gives us a unified program logic that can verify different progress properties using the same set of rules.
- We propose two novel mechanisms, *definite actions* and *stratified tokens*, to reason about blocking and delay, respectively. They are also our key techniques to avoid circularity in rely-guarantee style liveness reasoning. A definite action characterizes a thread's progress that does not rely on the progress of the environment. Each blocked thread waits for a queue of definite actions. Starvation-freedom requires the length of the queue be strictly decreasing, while deadlock-freedom allows *disciplined* queue jumps based on the token-transfer ideas [16, 24]. To avoid circular delay, we further generalize the token-transfer ideas by stratifying tokens into multiple levels, which enables us to verify complex algorithms that involve both nested locks and rollbacks (e.g., the optimistic list algorithm).
- By verifying linearizability and progress *together*, we can provide progress-aware abstractions for concurrent objects (see Sec. 5). Our logic is based on termination-preserving simulations as the meta-theory, which establish contextual refinements that assume fair scheduling at both the concrete and the abstract levels. We prove the contextual refinements are equivalent to linearizability and starvation-freedom/deadlock-freedom. The refinements allow us to replace object implementations with progress-aware abstract specifications when the client code is verified. As far as we know, our abstraction for deadlock-free (and linearizable) objects has never been proposed before.
- We have applied our logic to verify simple objects with coarse-grained synchronization using TAS locks, ticket locks [25] and various queue locks (including Anderson array-based locks, CLH locks and MCS locks) [13]. For examples with more permissive locking schemes, we have successfully verified the two-lock queues, and various fine-grained and optimistic list algorithms. To the best of our knowledge, we are the first to formally verify the starvation-freedom/deadlock-freedom of lock-coupling lists, optimistic lists and lazy lists.

Notice that with the assumption of fair scheduling, wait-freedom and lock-freedom are equivalent to starvation-freedom and deadlock-freedom, respectively. Therefore our logic can also be applied to verify wait-free and lock-free algorithms. We discuss this in Sec. 6.

In the rest of this paper, we first analyze the challenges and explain our approach informally in Sec. 2. Then we give the basic technical setting in Sec. 3, and present our logic in Sec. 4, whose soundness theorem, together with the abstraction theorem, is given in Sec. 5. We discuss how our logic supports wait-free and lock-free objects too in Sec. 6. Finally, we summarize the examples we have verified in Sec. 7, and discuss related work and conclude in Sec. 8.

(a) abstract operation `INC`: `<x++>;`

```
dfInc :
local b := false, r;
while (!b) { b := cas(&L, 0, cid); }  // lock L
r := x;  x := r + 1;  // critical section
L := 0;  // unlock L
```

(b) deadlock-free implementation `dfInc` using a test-and-set lock

```
sfInc :
local i, o, r;
i := getAndInc(next);
o := owner;  while (i != o) { o := owner; }
r := x;  x := r + 1;  // critical section
owner := i + 1;
```

(c) starvation-free implementation `sfInc` using a ticket lock

Figure 1. Counters.

2. Informal Development

Below we first give an overview of the traditional rely-guarantee logic for safety proofs [18], and the way to encode linearizability verification in the logic. Then we explain the challenges and our ideas in supporting liveness verification under fair scheduling.

2.1 Background

Rely-guarantee reasoning. In rely-guarantee reasoning [18], each thread is verified in isolation under some assumptions on its environment (i.e., the other threads in the system). The judgment is in the form of $R, G \vdash \{P\}C\{Q\}$, where the pre- and post-conditions P and Q specify the initial and final states respectively. The rely condition R specifies the assumptions on the environment, which are the permitted state transitions that the environment threads may have. The guarantee condition G specifies the possible transitions made by the thread itself. To ensure that parallel threads can collaborate, the guarantee of each thread needs to satisfy the rely of every other thread.

Encoding linearizability verification. As a working example, Fig. 1(b) shows a counter object `dfInc` implemented with a test-and-set (TAS) lock `L`. Verifying linearizability of `dfInc` requires us to prove it has the same abstract behaviors as `INC` in Fig. 1(a), which increments the counter `x` atomically.

Following previous work [21, 24, 31], one can extend a rely-guarantee logic to verify linearizability. We use an assertion $\mathsf{arem}(C)$ to specify as an auxiliary state the abstract operation C to be fulfilled, and logically execute C at the linearization point (LP) of the concrete implementation. For `dfInc`, we prove a judgment in the form of $R, G \vdash \{P \wedge \mathsf{arem}(\texttt{INC})\}\texttt{dfInc}\{Q \wedge \mathsf{arem}(\texttt{skip})\}$. Here R and G specify the object's actions (i.e., lock acquire and release, and the counter updates at both the concrete and the abstract sides) made by the environment and the current thread respectively. P and Q are relational assertions specifying the consistency relation between the program states at the concrete and the abstract sides. The postcondition $\mathsf{arem}(\texttt{skip})$ shows that at the end of `dfInc` there is no abstract operation to fulfill.

2.2 Challenges of Progress Verification

Progress properties of objects such as deadlock-freedom and starvation-freedom have various termination requirements of object methods. They must be satisfied with interference from other threads considered, which makes the verification challenging.

2.2.1 Non-Termination Caused by Interference

In a concurrent setting, an object method may fail to terminate due to interference from its environment. Below we point out there are two different kinds of interference that may cause thread non-termination, namely *blocking* and *delay*. Let's first see a classic deadlocking example.

```
DL-12 :                       DL-21 :
  lock L1; lock L2;             lock L2; lock L1;
  unlock L2; unlock L1;         unlock L1; unlock L2;
```

The methods DL-12 and DL-21 may fail to terminate because of the circular dependency of locks. This non-termination is caused by permanent *blocking*. That is, when DL-12 tries to acquire L2, it could be blocked if the lock has been acquired by DL-21.

For a second example, the call of the dfInc method (in Fig. 1(b)) by the left thread below may never terminate.

```
dfInc(); || while (true) dfInc();
```

When the left thread tries to acquire the lock, even if the lock is available at that time, the thread could be preempted by the right thread, who gets the lock ahead of the left. Then the left thread would fail at the cas command in the code of dfInc and have to loop at least one more round before termination. This may happen infinitely many times, causing non-termination of the dfInc method on the left. In this case we say the progress of the left method is *delayed* by its environment's successful acquirement of the lock.

The key difference between blocking and delay is that blocking is caused by the *absence* of certain environment actions, e.g., releasing a lock, while delay is caused by the *occurrence* of certain environment actions, e.g., acquiring the lock needed by the current thread (even if the lock is subsequently released). In other words, a blocked thread can progress only if its environment progresses first, while a delayed thread can progress if we suspend the execution of its environment.

Lock-free algorithms disallow blocking (thus they do not rely on fair scheduling), although delay is common, especially in optimistic algorithms. Starvation-free algorithms allow (limited) blocking, but not delay. As the dfInc example shows, delay from non-terminating clients may cause starvation. Deadlock-free algorithms allow both (with restrictions). As the optimistic list in Fig. 2(a) (explained in Sec. 2.3.4) shows, blocking and delay can be intertwined by the combined use of blocking-based synchronization and optimistic concurrency, which makes the reasoning significantly more challenging than reasoning about lock-free algorithms.

How do we come up with general principles to allow blocking and/or delay, but on the other hand to guarantee starvation-freedom or deadlock-freedom?

2.2.2 Avoid Circular Reasoning

Rely-guarantee style logics provide the power of thread-modular verification by *circular reasoning*. When proving the behaviors of a thread t guarantee G, we assume that the behaviors of the environment thread t$'$ satisfy R. Conversely, the proof of thread t$'$ relies on the assumptions on the behaviors of thread t.

However, circular reasoning is usually unsound in liveness verification [1]. For instance, we could prove termination of each thread in the deadlocking example above, under the assumption that each environment thread eventually releases the lock it owns. How do we avoid the circular reasoning without sacrificing rely-guarantee style thread-modular reasoning?

The deadlocking example shows that we should avoid circular reasoning to rule out circular dependency caused by blocking. Delay may also cause circular dependency too. Figure 2(b) shows a thread t using two locks. It first acquires L1 (line 1) and then tests whether L2 is available (line 2). If the test fails, the thread rolls back. It releases L1 (line 4), and then repeats the process of acquiring L1 (line 5) and testing L2 (line 6). Suppose another thread t$'$ does the opposite: repeatedly acquiring L2 and testing L1. In this example the acquirement of L2 by t$'$ may cause t to fail its test of the availability of L2. The test could have succeeded if t$'$ did not interfere, so t$'$ delays t. Conversely, the acquirement of L1 by t may delay t$'$. Then the two threads can cause each other to continually roll back, and neither method progresses.

Usually when delay is allowed, we need to make sure that the action delaying other threads is a "good" one in that it makes the executing thread progress (e.g., a step towards termination). This is the case with the "benign delays" in the dfInc example and the optimistic list example. But how do we tell if an action is good or not? The acquirements of locks in Fig. 2(b) do seem to be good because they make the threads progress towards termination. How do we prevent such lock acquirements from delaying others, which may cause circular delay?

2.2.3 Ad-Hoc Synchronization and Dynamic Locks

One may argue that the circularity can be avoided by simply enforcing certain orders of lock acquirements, which has been a standard way to avoid "deadlock cycles" (note this is a safety property, as we explained in Sec. 1). Although lock orders can help liveness reasoning, it has many limitations in practice.

First, the approach cannot apply for ad-hoc synchronization. For instance, there are no locks in the following deadlocking program.

```
x := 1;                ||  y := 1;
while (y = 1) skip;    ||  while (x = 1) skip;
x := 0;                ||  y := 0;
```

Moreover, sometimes we need to look into the lock implementation to prove starvation-freedom. For instance, the dfInc in Fig. 1(b) using a TAS lock is deadlock-free but not starvation-free. If we replace the TAS lock with a ticket lock, as in sfInc in Fig. 1(c), the counter becomes starvation-free. Again, there are actually no locks in the programs if we have to work at a low abstraction level to look into lock implementations.

Second, it can be difficult to enforce the ordering for fine-grained algorithms on dynamic data structures (e.g., lock-coupling list). Since the data structure is changing dynamically, the set of locks associated with the nodes is dynamic too. To allow a thread to determine dynamically the order of locks, we have to ensure its view of ordering is consistent with all the other threads in the system, a challenge for thread-modular verification. Although dynamic locks are supported in some previous work treating deadlock-freedom as a safety property (e.g., [4, 19]), it is unclear how to apply the techniques for general progress reasoning, with possible combination of locks, ad-hoc synchronization and rollbacks.

2.3 Our Approaches

To address these problems, our logic enforces the following principles to permit restricted forms of blocking and delay, but prevent circular reasoning and non-termination.

First, if a thread is blocked, the events it waits for must eventually occur. To avoid circular reasoning, we find "definite actions" of each thread, which under fair scheduling will definitely happen once enabled, regardless of the interference from the environment. Then each blocked thread needs to show it waits for only a finite number of definite actions from the environment threads. They form an acyclic queue, and there is always at least one of them enabled. This is what we call "definite progress", which is crucial for proving starvation-freedom.

Second, actions of a thread can delay others *only if* they are making the executing object method to move towards termination. Each object method can only execute a finite number of such delaying actions to avoid indefinite delay. This is enforced by assigning a finite number of tokens to each method. A token must be paid to execute a delaying action.

Third, we divide actions of a thread into normal ones (which do not delay others) and delaying ones, and further stratify delaying actions into multiple levels. When a thread is delayed by a level-k action from its environment, it is allowed to execute not only more normal actions, but also more delaying actions at lower levels. Allowing one delaying action to trigger more steps of other delaying actions is necessary for verifying algorithms with nested locks and rollbacks, such as the optimistic lists in Fig. 2(a). The stratification prevents the circular delay in the example of Fig. 2(b).

Fourth, our delaying actions and definite actions are all semantically specified as part of object specifications, therefore we can support ad-hoc synchronizaiton and do not rely on built-in synchronization primitives to enforce ordering of events. Moreover, since the specifications are all parametrized over states, they are expressive enough to support dynamic locks as in lock-coupling lists. Also our "definite progress" condition allows each blocked thread to decide *locally* and *dynamically* a queue of definite actions it waits for. There is no need to enforce a global ordering of blocking dependencies agreed by every thread. This also provides thread-modular support of dynamic locks.

Below we give more details about some of these key ideas.

2.3.1 Using Tokens to Prevent Infinite Loops

The key to ensuring termination is to require each loop to terminate. Earlier work [16, 24] requires each round of the loop to consume resources called tokens. The rule for loops is in the following form:

$$\frac{P \wedge B \Rightarrow P' * \Diamond \qquad R, G \vdash \{P'\}C\{P\}}{R, G \vdash \{P\}\textbf{while}\ (B)\ C\{P \wedge \neg B\}}\ (\text{TERM})$$

Here $\Diamond$ represents one token, and "$*$" is the normal separating conjunction in separation logic. The premise says the precondition P' of the loop body C has one less token than P, showing that one token needs to be consumed to start this new round of loop. Since the number of tokens strictly decreases, we know the loop must terminate when the thread has no token.

We use this simple idea to enforce termination of loops, and extend it to handle blocking and delay in a concurrent setting.

2.3.2 Definite Actions and Definite Progress

Our approach to cut the blocking-caused circular dependency is inspired by the implementation of ticket locks, which is used to implement the starvation-free counter `sfInc` in Fig. 1(c). It uses the shared variables `owner` and `next` to guarantee the first-come-first-served property of the lock. Initially `owner` equals `next`. To acquire the lock, a thread atomically increments `next` and reads its old value to a variable `i` (line 2). The value of `i` becomes the thread's ticket. The thread waits until `owner` equals its ticket value `i` (line 3). Finally the lock is released by incrementing `owner` (line 5) such that the next waiting thread (the thread with ticket $\mathtt{i}+1$, if there is one) can now enter the critical section.

We can see `sfInc` is not concerned with the circular dependency problem. Intuitively the ticket lock algorithm ensures that the threads requesting the lock always constitute a queue $\mathtt{t}_1, \mathtt{t}_2, \ldots, \mathtt{t}_n$. The head thread, $\mathtt{t}_1$, gets the ticket number which equals `owner` and can immediately acquire the lock. Once it releases the lock (by increasing `owner`), $\mathtt{t}_1$ is dequeued. Moreover, for any thread `t` in this queue, the number of threads ahead of `t` never increases. Thus `t` must eventually become the head of the queue and acquire the lock. Here the dependencies among progress of the threads are in concert with the queue.

Following this queue principle, we explicitly specify the queue of progress dependencies in our logic to avoid circular reasoning.

Definite actions. First, we introduce a novel notion called a "definite action" $\mathcal{D}$, which models a thread action that, once enabled, must be eventually finished regardless of what the environment does. In detail, $\mathcal{D}$ is in the form of $P_d \leadsto Q_d$. It requires in every execution that Q_d should eventually hold *if* P_d holds, and P_d should be preserved (by both the current thread and the environment) until Q_d holds. For `sfInc`, the definite action $P_d \leadsto Q_d$ of a thread can be defined as follows. P_d says that `owner` equals the thread's ticket number `i`, and Q_d says that `owner` has been increased to $\mathtt{i}+1$. That is, a thread definitely releases the lock when acquiring it. Of course we have to ensure in our logic that $\mathcal{D}$ is indeed definite. We will explain in detail the logic rule that enforces it in Sec. 4.2.2.

Definite progress. Then we use definite actions to prove termination of loops. We need to first find an assertion Q specifying the condition when the thread `t` can progress on its own, i.e., it is *not* blocked. Then we enforce the following principles:

1. If Q is continuously true, we need to prove the loop terminates following the idea of the TERM rule;
2. If Q is false, the following must *always* be true:
   (a) There is a finite queue of definite actions of other threads that the thread `t` is waiting for, among which there is at least one (from a certain thread $\mathtt{t}'$) enabled. The length of the queue is E.
   (b) E decreases whenever one of these definite actions is finished;
   (c) The expression E is never increased by any threads (no matter whether Q holds or not); and it is non-negative.

We can see E serves as a well-founded metric. By induction over E we know eventually Q holds, which implies the termination of the loop by the above condition 1.

These conditions are enforced in our new inference rule for loops, which extends the TERM rule (in Sec. 2.3.1) and is presented in Sec. 4.2.2. The condition 2 shows the use of definite actions in our reasoning about progress. We call it the "definite progress" condition.

The reasoning above implicitly makes use of the fairness assumption. The fair scheduling ensures that the environment thread $\mathtt{t}'$ mentioned in the condition 2(a) is scheduled infinitely often, therefore its definite action will definitely happen. By conditions 2(b) and 2(c) we know E will become smaller. In this way E keeps decreasing until Q holds eventually.

For `sfInc`, Q is defined as $(\mathtt{i} = \mathtt{owner})$ and the metric E is $(\mathtt{i} - \mathtt{owner})$. Whenever an environment thread $\mathtt{t}'$ finishes a definite action by releasing the lock, it increases `owner`, so E decreases. When E is decreased to 0, the current thread is unblocked. Its loop terminates and it succeeds in acquiring the lock.

2.3.3 Allowing Queue Jumps for Deadlock-Free Objects

The method `dfInc` in Fig. 1(b) implements a deadlock-free counter using the TAS lock. If the current thread `t` waits for the lock, we know the queue of definite actions it waits for is of length *one* because it is possible for the thread to acquire the lock immediately after the lock is released. However, as we explain in Sec. 2.2.1, another thread $\mathtt{t}'$ may preempt `t` and do a successful `cas`. Then thread `t` is blocked and waits for a queue of definite actions again. This delay caused by thread $\mathtt{t}'$ can be viewed as a queue jump in our definite-progress-based reasoning. Actually `dfInc` cannot satisfy the definite progress requirement because we cannot find a strictly decreasing queue size E. It is not starvation-free.

However, the queue jump here is acceptable when verifying deadlock-freedom. This is because thread $\mathtt{t}'$ delays `t` only if $\mathtt{t}'$

```
 1  local b := false, p, c;
 2  while (!b) {
 3    (p, c) := find(e);
 4    lock p; lock c;
 5    b := validate(p, c);
 6    if (!b) {
 7      unlock c; unlock p; }
 8  }
 9  update(p, c, e);
10  unlock c; unlock p;
```
(a) optimistic list

```
 1  lock L1;
 2  local r := L2;
 3  while (r != 0) {
 4    unlock L1;
 5    lock L1;
 6    r := L2;
 7  }
 8  lock L2;
 9  unlock L2;
10  unlock L1;
```
(b) rollback

Figure 2. Examples with multiple locks.

successfully acquires the lock, which allows it to eventually finish the `dfInc` method. Thus the system as a whole progresses.

Nevertheless, as explained in Sec. 2.2.2, we have to make sure the queue jump (which is a special form of delay) is a "good" one.

We follow the token-transfer ideas [16, 24] to support disciplined queue jumps. We explicitly specify in the rely/guarantee conditions which steps could delay the progress of other threads (jump their queues). To prohibit unlimited queue jumps without making progress, we assign a finite number m of $\blacklozenge$-tokens to an object method, and require that a thread can do at most m delaying actions before the method finishes.

Whenever a step of thread $\mathtt{t}'$ delays the progress of thread $\mathtt{t}$, we require $\mathtt{t}'$ to consume one $\blacklozenge$-token. At the same time, thread $\mathtt{t}$ could increase $\lozenge$-tokens so that it can loop more rounds. Besides, we redefine the definite progress condition to allow the metric E (about the length of the queue) to be increased when an environment thread jumps the queue at the cost of a $\blacklozenge$-token.

2.3.4 Allowing Rollbacks for Optimistic Locking

The ideas we just explained already support simple deadlock-free objects such as `dfInc` in Fig. 1(b), but they cannot be applied to objects with optimistic synchronization, such as optimistic lists [13] and lazy lists [11].

Figure 2(a) shows part of the optimistic list implementation. Each node of the list is associated with a TAS lock, the same lock as in Fig. 1(b). A thread first traverses the list without acquiring any locks (line 3). The traversal `find` returns two adjacent node pointers `p` and `c`. The thread then locks the two nodes (line 4), and calls `validate` to check if the two nodes are still valid list nodes (line 5). If validation succeeds, then the thread performs its updates (adding or removing elements) to the list (line 9). Otherwise it releases the two node locks (line 7) and restarts the traversal.

For this object, when the validation fails, a thread will release the locks it has acquired and roll back. Thus the thread may acquire the locks for an unbounded number of times. Since each lock acquirement will delay other threads requesting the same lock and each delaying action should consume one $\blacklozenge$-token, it seems that the thread would need an infinite number of $\blacklozenge$-tokens, which we prohibit in the preceding subsection to ensure deadlock-freedom, even though this list object is indeed deadlock-free.

We generalize the token-transfer ideas to allow rollbacks in order to verify this kind of optimistic algorithms, but still have to be careful to avoid the circular delay caused by the "bad" rollbacks in Fig. 2(b), as we explain in Sec. 2.2.2.

Our solution is to stratify the delaying actions. Each action is now labeled with a level k. The normal actions which cannot delay other threads are at the lowest level 0. The $\blacklozenge$-tokens are stratified accordingly. A thread can roll back and do more actions at level k only when its environment does an action at a higher level k', at the cost of a k'-level $\blacklozenge$-token. Note that the $\blacklozenge$-tokens at the highest level are strictly decreasing, which means a thread cannot roll back its actions of the highest level. In fact, the numbers of $\blacklozenge$-tokens at all levels constitute a tuple $(m_k, \ldots, m_2, m_1)$. It is strictly descending along the dictionary order.

$$
\begin{array}{llcl}
(\mathit{MName}) & f & \in & \mathit{String} \qquad\qquad (\mathit{ThrdID})\ \ \mathtt{t} \ \in\ \mathit{Nat} \\
(\mathit{Expr}) & E & ::= & x \mid n \mid E+E \mid \ldots \\
(\mathit{BExp}) & B & ::= & \mathsf{true} \mid \mathsf{false} \mid E=E \mid !B \mid \ldots \\
(\mathit{Instr}) & c & ::= & x := E \mid x := [E] \mid [E] := E \mid \mathbf{print}(E) \mid \ldots \\
(\mathit{Stmt}) & C & ::= & \mathbf{skip} \mid c \mid x := f(E) \mid \mathbf{return}\, E \mid \langle C \rangle \\
 & & \mid & C;C \mid \mathbf{if}\,(B)\,C\,\mathbf{else}\,C \mid \mathbf{while}\,(B)\{C\} \\
(\mathit{Prog}) & W & ::= & \mathbf{skip} \mid \mathbf{let}\,\Pi\,\mathbf{in}\,C \,\|\, \ldots \,\|\, C \\
(\mathit{ODecl}) & \Pi,\Gamma & ::= & \{f_1 \rightsquigarrow (x_1, C_1), \ldots, f_n \rightsquigarrow (x_n, C_n)\} \\
 & & & \\
(\mathit{Store}) & s, \mathbb{s} & \in & \mathit{Var} \rightharpoonup \mathit{Int} \qquad (\mathit{Heap})\ \ h, \mathbb{h} \ \in\ \mathit{Nat} \rightharpoonup \mathit{Int} \\
(\mathit{Mem}) & \sigma, \Sigma & ::= & (s, h) \qquad\qquad (\mathit{PState})\ \ \mathcal{S} \ ::=\ (\sigma_c, \sigma_o, \ldots)
\end{array}
$$

Figure 3. The language syntax and state model.

The stratified $\blacklozenge$-tokens naturally prohibit the circular delay problem discussed in Sec. 2.2.2 with the object in Fig. 2(b) . The deadlock-free optimistic list in Fig. 2(a) can now be verified. We classify its delaying actions into two levels. Actions at level 2 (the highest level) update the list, which correspond to line 9 in Fig. 2(a), and each method can do only *one* such action. Lock acquirements are classified at level 1, so a thread is allowed to roll back and re-acquire the locks when its environment updates the list.

3. Programming Language

Figure 3 gives the syntax of the language. A program W consists of multiple parallel threads sharing an object Π. We say the threads are the *clients* of the object. An object declaration Π is a mapping from a method name f to a pair of argument and code (method body). The statements C are similar to those in the simple language used for separation logic. The command $\mathbf{print}(E)$ generates externally observable events, which allows us to observe behaviors of programs. We assume it is used only in the code of clients. We use $\langle C \rangle$ to represent an atomic block in which C is executed atomically.

To simplify the presentation, we make several simplifications here. We assume there is only one concurrent object in the system. Each method of the object takes only one argument, and the method body ends with a **return** E command. Also we assume there is no nested method invocation. For the abstract object specification Γ, each method body is an atomic operation $\langle C \rangle$ (ahead of the **return** command), and executing the code is always safe and never blocks.

The model of program states $\mathcal{S}$ is defined at the bottom of Fig. 3. To ensure that the client code does not touch the object state, in $\mathcal{S}$ we separate the states accessed by clients (σ_c) and by the object (σ_o). Here $\mathcal{S}$ may also contain auxiliary data about the control stacks for method calls. Execution of programs is modeled as a labeled transition system $(W, \mathcal{S}) \overset{e}{\longmapsto} (W', \mathcal{S}')$. Here e is an event (either observable or not) produced by the transition. We give the small-step transition semantics in TR [22].

Below we often write Σ, $\mathbb{s}$ and $\mathbb{h}$ for the notations at the abstract level to distinguish from the concrete-level ones (σ, s and h).

4. Program Logic LiLi

LiLi verifies the linearizability of objects by proving the method implementations refine abstract atomic operations. The top level judgment is in the form of $\mathcal{D}, R, G \vdash \{P\}\Pi \preceq \Gamma$. (The OBJ rule for this judgment is given in Fig. 7 and will be explained later.) To verify an object Π, we give a corresponding object specification Γ (see Fig. 3), which maps method names to atomic commands. In addition, we need to provide an object invariant (P) and rely/guarantee conditions (R and G) for the refinement reasoning in a concurrent

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

Figure 4. Syntax of the assertion language.

setting. Here P is a relational assertion that specifies the consistency relation between the concrete data representation and the abstract value. Similarly, R and G lift regular rely and guarantee conditions to the binary setting, which now specify transitions of states at both the concrete level and the abstract level. The definite action $\mathcal{D}$ is a special form of state transitions used for *progress* reasoning. The definitions of P, R, G and $\mathcal{D}$ are shown in Sec. 4.1.

Note LiLi is a logic for concurrent objects Π only. We do not provide logic rules for clients. See Sec. 5 for more discussions.

To simplify the presentation in this paper, we describe LiLi based on the plain Rely-Guarantee Logic [18]. Also, to avoid "variables as resources" [27], we assume program variables are either thread-local or read-only. The full version of LiLi (see TR [22]) extends the more advanced Rely-Guarantee-based logic LRG [7] to support dynamic allocation and ownership transfer. It also drops the assumption about program variables.

4.1 Assertions

We define assertions in Fig. 4. The relational state assertions P and Q specify the relationship between the concrete state σ and the abstract state Σ. Here we use $\mathbb{s}$ and $\mathbb{h}$ for the store and the heap at the abstract level (see Fig. 3). For simplicity, we assume the program variables used in the concrete code are different from those in the abstract code (e.g., we use x and X at the concrete and abstract levels respectively). We use the relational state $\mathfrak{S}$ to represent the pair of states (σ, Σ), as defined in Fig. 5.

Figure 5(a) defines semantics of state assertions. The boolean expression B holds if it evaluates to true at the combined store of s and $\mathbb{s}$. emp describes empty heaps. The assertion $E_1 \mapsto E_2$ specifies a singleton heap at the concrete level with the value of the expression E_2 stored at the location E_1. Its counterpart for an abstract level heap is represented as $E_1 \Mapsto E_2$. Semantics of separating conjunction $P * Q$ is similar as in separation logic, except that it is now lifted to relational states $\mathfrak{S}$. The disjoint union of two relational states is defined at the top of the figure. Semantics of the assertion $\lfloor\!\lfloor p \rfloor\!\rfloor$ will be defined latter (see Fig. 5(c)).

Rely/guarantee assertions R and G specify the transitions over the relational states $\mathfrak{S}$. Their semantics is defined in Fig. 5(b). The action $P \ltimes_k Q$ says that the initial relational states satisfy P and the resulting states satisfy Q. We can ignore the index k for now, which is used to stratify actions that may delay the progress of other threads and will be explained in Sec. 4.3. $[P]$ specifies identity transitions with the initial states satisfying P. Semantics of $\lfloor G \rfloor_0$ is defined in Sec. 4.3 too (see Fig. 10). Below we use $P \ltimes Q$ as a shorthand for $P \ltimes_0 Q$. We also use Id for [true], which represents arbitrary identity transitions.

We further instrument the relational state assertions with the specifications of the abstract level code and various tokens. The resulting *full assertions* p and q are defined in Fig. 4, whose semantics is given in Fig. 5(c). The assertion p is interpreted over $(\mathfrak{S}, (u, w), C)$. C is the abstract-level code that remains to be refined. It is specified by the assertion $\mathsf{arem}(C)$. Since our logic verifies linearizability of objects, C is always in the form of atomic commands $\langle C' \rangle$ (ahead of **return** commands). The pair

$$\mathfrak{S} ::= (\sigma, \Sigma) \qquad (\sigma, \Sigma) \uplus (\sigma', \Sigma') \overset{\text{def}}{=} (\sigma \uplus \sigma', \Sigma \uplus \Sigma')$$
$$\text{where } (s, h) \uplus (s', h') \overset{\text{def}}{=} (s, h \uplus h'), \text{ if } s = s'$$

$$\begin{array}{lll}
((s, h), (\mathbb{s}, \mathbb{h})) \models B & \text{iff} & [\![B]\!]_{s \uplus \mathbb{s}} = \mathbf{true} \\
((s, h), (\mathbb{s}, \mathbb{h})) \models \mathsf{emp} & \text{iff} & \mathit{dom}(h) = \mathit{dom}(\mathbb{h}) = \emptyset \\
((s, h), (\mathbb{s}, \mathbb{h})) \models E_1 \mapsto E_2 & \text{iff} & h = \{[\![E_1]\!]_{s \uplus \mathbb{s}} \leadsto [\![E_2]\!]_{s \uplus \mathbb{s}}\} \\
((s, h), (\mathbb{s}, \mathbb{h})) \models E_1 \Mapsto E_2 & \text{iff} & \mathbb{h} = \{[\![E_1]\!]_{s \uplus \mathbb{s}} \leadsto [\![E_2]\!]_{s \uplus \mathbb{s}}\} \\
\mathfrak{S} \models P * Q & \text{iff} & \exists \mathfrak{S}_1, \mathfrak{S}_2.\, \mathfrak{S} = \mathfrak{S}_1 \uplus \mathfrak{S}_2 \\
 & & \wedge (\mathfrak{S}_1 \models P) \wedge (\mathfrak{S}_2 \models Q)
\end{array}$$

(a) semantics of relational state assertions P and Q

$$\begin{array}{lll}
(\mathfrak{S}, \mathfrak{S}') \models P \ltimes_k Q & \text{iff} & (\mathfrak{S} \models P) \wedge (\mathfrak{S}' \models Q) \\
(\mathfrak{S}, \mathfrak{S}') \models [P] & \text{iff} & (\mathfrak{S}' = \mathfrak{S}) \wedge (\mathfrak{S} \models P)
\end{array}$$

(b) semantics of relational rely/guarantee assertions R and G

$$\begin{array}{lll}
(\mathfrak{S}, (u, w), C) \models P & \text{iff} & \mathfrak{S} \models P \\
(\mathfrak{S}, (u, w), C) \models \mathsf{arem}(C') & \text{iff} & C = C' \\
(\mathfrak{S}, (u, w), C) \models \Diamond(E) & \text{iff} & \exists n.\, ([\![E]\!]_{\mathfrak{S}.s} = n) \wedge (n \leq w) \\
(\mathfrak{S}, (u, w), C) \models \blacklozenge(E_k, \ldots, E_1) & \text{iff} & ([\![E_k]\!]_{\mathfrak{S}.s}, \ldots, [\![E_1]\!]_{\mathfrak{S}.s}) \leq u \\
(\mathfrak{S}, (u, w), C) \models \lfloor p \rfloor_\Diamond & \text{iff} & \exists w'.\, (\mathfrak{S}, (u, w'), C) \models p \\
\mathfrak{S} \models \lfloor\!\lfloor p \rfloor\!\rfloor & \text{iff} & \exists u, w, C.\, (\mathfrak{S}, (u, w), C) \models p
\end{array}$$

$$C \uplus C' \overset{\text{def}}{=} \begin{cases} C' & \text{if } C = \mathbf{skip} \\ C & \text{if } C' = \mathbf{skip} \end{cases}$$

$$(\mathfrak{S}, (u, w), C) \uplus (\mathfrak{S}', (u', w'), C') \overset{\text{def}}{=} (\mathfrak{S} \uplus \mathfrak{S}', (u + u', w + w'), C \uplus C')$$

(c) semantics of full assertions p and q

Figure 5. Semantics of assertions.

(u, w) records the numbers of various tokens $\blacklozenge$ and $\Diamond$. It serves as a well-founded metric for our progress reasoning. Informally w specifies the upper bound of the number of loop rounds that the current thread can execute if it is neither blocked nor delayed by its environment. The assertion $\Diamond(E)$ says the number w of $\Diamond$-tokens is *no less than* E. Therefore $\Diamond(0)$ always holds, and $\Diamond(n+1)$ implies $\Diamond(n)$ for any n. We postpone the explanation of u and the assertion $\blacklozenge(E_k, \ldots, E_1)$ to Sec. 4.3. Below we use $\Diamond$ as the shorthand for $\Diamond(1)$. We use $\lfloor p \rfloor_\Diamond$ to ignore the descriptions in p about the number of $\Diamond$-tokens. $\lfloor\!\lfloor p \rfloor\!\rfloor$ converts p back to a relational state assertion.

Separating conjunction $p * q$ has the standard meaning as in separation logic, which says p and q hold over disjoint parts of $(\mathfrak{S}, (u, w), C)$ respectively (the formal definition elided here). The disjoint union is defined in Fig. 5(c). The disjoint union of the numbers of tokens is the sum of them. The disjoint union of C_1 and C_2 is defined only if at least one of them is **skip**. Therefore we know the following holds (for any P and C):

$$(P \wedge \mathsf{arem}(C) \wedge \Diamond) * (\Diamond \wedge \mathsf{emp}) \Leftrightarrow (P \wedge \mathsf{arem}(C) \wedge \Diamond(2))$$

Definite actions. Fig. 4 also defines definite actions $\mathcal{D}$, which can be treated as special forms of rely/guarantee conditions. Their semantics is given in Fig. 6(a). $P \leadsto Q$ specifies the transitions where the final states satisfy Q *if* the initial states satisfy P. It is different from $P \ltimes Q$ in that $P \leadsto Q$ does not restrict the transitions if initially P does not hold. Consider the following example $\mathcal{D}_x$.

$$\mathcal{D}_x \overset{\text{def}}{=} \forall n.\, ((\mathtt{x} \mapsto n) \wedge (n > 0)) \leadsto (\mathtt{x} \mapsto n + 1)$$

$\mathcal{D}_x$ describes the state transitions which increment x if x is positive initially. It is satisfied by any transitions where initially x is not positive. The conjunction $\mathcal{D}_1 \wedge \mathcal{D}_2$ is useful for enumerating definite actions. For instance, when the program uses two locks L1 and L2,

$$(\mathfrak{S},\mathfrak{S}') \models P \rightsquigarrow Q \quad \text{iff} \quad (\mathfrak{S} \models P) \implies (\mathfrak{S}' \models Q)$$
$$(\mathfrak{S},\mathfrak{S}') \models \forall x.\, \mathcal{D} \quad \text{iff} \quad \forall n.\, (\mathfrak{S}\{x \rightsquigarrow n\}, \mathfrak{S}'\{x \rightsquigarrow n\}) \models \mathcal{D}$$
$$(\mathfrak{S},\mathfrak{S}') \models \mathcal{D}_1 \wedge \mathcal{D}_2 \quad \text{iff} \quad ((\mathfrak{S},\mathfrak{S}') \models \mathcal{D}_1) \wedge ((\mathfrak{S},\mathfrak{S}') \models \mathcal{D}_2)$$

(a) semantics of $\mathcal{D}$

$$\begin{aligned}
\mathsf{Enabled}(P \rightsquigarrow Q) &\overset{\text{def}}{=} P\\
\mathsf{Enabled}(\forall x.\, \mathcal{D}) &\overset{\text{def}}{=} \exists x.\, \mathsf{Enabled}(\mathcal{D})\\
\mathsf{Enabled}(\mathcal{D}_1 \wedge \mathcal{D}_2) &\overset{\text{def}}{=} \mathsf{Enabled}(\mathcal{D}_1) \vee \mathsf{Enabled}(\mathcal{D}_2)\\
\langle \mathcal{D} \rangle &\overset{\text{def}}{=} \mathcal{D} \wedge (\mathsf{Enabled}(\mathcal{D}) \ltimes \mathsf{true})\\
[\mathcal{D}] &\overset{\text{def}}{=} \mathsf{Enabled}(\mathcal{D}) \rightsquigarrow \mathsf{Enabled}(\mathcal{D})
\end{aligned}$$

$$\mathcal{D}' \leqslant \mathcal{D} \text{ iff } (\mathsf{Enabled}(\mathcal{D}') \Rightarrow \mathsf{Enabled}(\mathcal{D})) \wedge (\mathcal{D} \Rightarrow \mathcal{D}')$$

(b) useful syntactic sugars

Figure 6. Semantics of definite actions.

the definite action $\mathcal{D}$ of the whole program is usually in the form of $\mathcal{D}_1 \wedge \mathcal{D}_2$, where $\mathcal{D}_1$ and $\mathcal{D}_2$ specify `L1` and `L2` respectively.

We define some useful syntactic sugars in Fig. 6(b). The state assertion $\mathsf{Enabled}(\mathcal{D})$ takes the guard condition of $\mathcal{D}$. We use $\langle \mathcal{D} \rangle$ to represent the state transitions of $\mathcal{D}$ when it is enabled at the initial state. Intuitively $\langle \mathcal{D} \rangle$ gives us the corresponding "$\ltimes$" actions. For instance, $\langle P \rightsquigarrow Q \rangle$ is equivalent to $P \ltimes Q$. For the example $\mathcal{D}_x$ defined above, $\langle \mathcal{D}_x \rangle$ is equivalent to the following:

$$\exists n.\, ((\mathtt{x} \mapsto n) \wedge (n > 0)) \ltimes (\mathtt{x} \mapsto n + 1)$$

In addition, we define the syntactic sugar $[\mathcal{D}]$ as a definite action describing the preservation of $\mathsf{Enabled}(\mathcal{D})$. For the example $\mathcal{D}_x$ above, $[\mathcal{D}_x]$ represents the following definite action:

$$(\exists n.\, (\mathtt{x} \mapsto n) \wedge (n > 0)) \rightsquigarrow (\exists n.\, (\mathtt{x} \mapsto n) \wedge (n > 0))$$

It specifies the transitions which keep `x` positive if it is positive initially. The notation $\mathcal{D}' \leqslant \mathcal{D}$ will be explained later in Sec. 4.2.2. Since $\mathcal{D}$ is a special rely/guarantee assertion, the semantics of $\mathcal{D} \Rightarrow \mathcal{D}'$ follows the standard meaning of $R \Rightarrow R'$ [7] (or see the definition in Fig. 10).

Thread IDs as implicit assertion parameters. All the assertions in our logic, including state assertions, rely/guarantee conditions and definite actions, are implicitly parametrized over thread IDs. Although our logic does modular reasoning about the object code without any knowledge about clients, it is useful for assertions to refer to thread IDs. For instance, we may use $\mathtt{L} \mapsto \mathtt{t}$ to represent that the lock `L` is acquired by the thread `t`. We use $P_{\mathtt{t}}$ to represent the instantiation of the thread ID parameter in P with `t`, which means P holds on thread `t`. Then P alone can also be understood as $\forall \mathtt{t}.P_{\mathtt{t}}$, and $P \Rightarrow Q$ can be viewed as $\forall \mathtt{t}.P_{\mathtt{t}} \Rightarrow Q_{\mathtt{t}}$. The same convention applies to rely/guarantee conditions and definite actions.

4.2 Verifying Starvation-Freedom with Definite Actions

Figure 7 presents the inference rules of LiLi. We explain the logic in two steps. In this subsection we explain the use of definite actions to reason about starvation-freedom. Then we explain the delay mechanism for deadlock-freedom in Sec. 4.3.

4.2.1 The OBJ Rule

The OBJ rule requires that each method in Π refine its atomic specification in Γ. Starting from the initial concrete and abstract object states related by P, and with the equivalent method arguments x and y at the concrete and the abstract levels, the method body C must fulfill the abstract atomic operation C'. We can temporarily ignore the assertion $\blacklozenge(E_k, \ldots, E_1)$ for deadlock-freedom.

The last three premises of the OBJ rule check the well-formedness of the specifications. The first one says the guarantee of one thread must imply the rely of all others, a standard requirement in rely/guarantee reasoning. In Fig. 8 we give a simplified definition of wffAct used in the second premise. Its complete definition is given in Fig. 11, which will be explained later when we introduce stratified actions and $\blacklozenge$-tokens. $\mathsf{wffAct}(R, \mathcal{D})$ says once a definite action $\mathcal{D}_{\mathtt{t}}$ of a thread `t` is enabled it cannot be disabled by an environment step in $R_{\mathtt{t}}$. Also such an environment step either fulfills a definite action $\mathcal{D}_{\mathtt{t}'}$ of some thread `t'` different from `t`, or preserves $\mathsf{Enabled}(\mathcal{D}_{\mathtt{t}'})$ too. Together with the previous premise $G_{\mathtt{t}'} \Rightarrow R_{\mathtt{t}}$, this condition implies $\forall \mathtt{t}'.\, G_{\mathtt{t}'} \Rightarrow [\mathcal{D}_{\mathtt{t}'}] \vee \mathcal{D}_{\mathtt{t}'}$. Therefore, once $\mathcal{D}_{\mathtt{t}}$ is enabled, the only way to disable it is to let the thread `t` finish the action. As an example, consider the following $\mathcal{D}_{\mathtt{t}}$:

$$\mathcal{D}_{\mathtt{t}} \overset{\text{def}}{=} (\mathtt{L} \mapsto \mathtt{t}) \rightsquigarrow (\mathtt{L} \mapsto 0)$$

It says that whenever a thread `t` acquires the lock `L`, it will finally release the lock. Then, $\mathsf{wffAct}(R, \mathcal{D})$ requires that when `t` acquires `L`, every step in the system either keeps `L` unchanged or releases `L`. In particular, $R_{\mathtt{t}}$ keeps `L` unchanged, that is, the environment cannot update the lock when $\mathtt{L} \mapsto \mathtt{t}$.

The last premise $(P \Rightarrow \neg\mathsf{Enabled}(\mathcal{D}))$ says there cannot be enabled but unfinished definite actions when the method terminates and the object invariant P is true.

The judgment $\mathcal{D}, R, G \vdash \{p \wedge \mathsf{arem}(C')\}C\{q \wedge \mathsf{arem}(\mathbf{skip})\}$ establishes a *simulation relation* between C and C', which ensures the preservation of termination when the environment guarantees the definite action $\mathcal{D}$. It also ensures the execution of C guarantees $\mathcal{D}$ too. We explain the key rules for the judgment below. The complete set of rules are presented in TR [22].

4.2.2 The WHL Rule for Loops

The WHL rule, shown in Fig. 7, is the most important rule of the logic. It establishes *both* of the following properties of the loop:

(1) it cannot loop forever with $\mathcal{D}$ continuously enabled;

(2) it cannot loop forever unless the current thread is waiting for some definite actions of its environment.

The former guarantees that a definite action of the current thread will *definitely* happen once it is enabled. The latter is crucial to establish the starvation-freedom.

Why definite actions are definite. The WHL verifies the loop body with a precondition p', which can be derived from the loop invariant p if B holds. Moreover, we require each iteration to consume a $\lozenge$-token if $\mathsf{Enabled}(\mathcal{D})$ holds at the beginning, as shown by the second premise (ignore the assertion Q for now). Since each thread has only a finite number of $\lozenge$-tokens, the loop must terminate if $\mathsf{Enabled}(\mathcal{D})$ is continuously true.

However, the last premise of the OBJ rule says $\mathsf{Enabled}(\mathcal{D})$ cannot be true when the method terminates. Therefore, $\mathsf{Enabled}(\mathcal{D})$ cannot be continuously true. Also recall the other two side conditions ($\mathsf{wffAct}(R, \mathcal{D})$ and $G_{\mathtt{t}} \Rightarrow R_{\mathtt{t}'}$) of the OBJ rule guarantee that, once $\mathsf{Enabled}(\mathcal{D})$ holds, the only way to make it false is to let the current thread finish the action.

Putting all these together, we know $\mathcal{D}$ will be finished once it is enabled, even with the interference R.

Starvation-freedom. To establish starvation-freedom, we need to find a condition Q saying the current thread is *not* blocked by others. Then the second premise requires each iteration to consume a $\lozenge$-token if Q holds at the beginning. Since the number of tokens is finite, the loop must terminate if Q always holds.

If Q is false, the current thread is blocked by others, and it does not need to consume $\lozenge$-tokens. Then the premise $(R, G\colon \mathcal{D}' \xrightarrow{f} Q)$

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

$$\frac{\begin{array}{c}p \wedge B \Rightarrow p' \qquad p \wedge B \wedge (\mathsf{Enabled}(\mathcal{D}) \vee Q) \Rightarrow p' * (\Diamond \wedge \mathsf{emp}) \qquad \mathcal{D}, R, G \vdash \{p'\}C\{p\} \\ p \Rightarrow J \qquad \mathsf{Sta}(J, R \vee G) \qquad \mathcal{D}' \leqslant \mathcal{D} \qquad \mathsf{wffAct}(R, \mathcal{D}') \qquad J \Rightarrow (R, G \colon \mathcal{D}' \xrightarrow{f} Q)\end{array}}{\mathcal{D}, R, G \vdash \{p\}\textbf{while}\ (B)\{C\}\{p \wedge \neg B\}} \ (\textsc{whl}) \qquad \frac{\mathcal{D}, R, G \vdash \{p\}C\{q\}}{\mathcal{D}, R, G \vdash \{\lfloor p \rfloor_{\Diamond}\}C\{\lfloor q \rfloor_{\Diamond}\}} \ (\textsc{hide-}\Diamond)$$

$$\frac{\vdash [p]C[q'] \qquad q' \Rrightarrow_k q \qquad (\llbracket p \rrbracket \ltimes_k \llbracket q \rrbracket) \Rightarrow G}{\mathcal{D}, \mathsf{Id}, G \vdash \{p\}\langle C \rangle\{q\}} \ (\textsc{atom}) \qquad \frac{\mathcal{D}, \mathsf{Id}, G \vdash \{p\}\langle C \rangle\{q\} \qquad \mathsf{Sta}(\{p, q\}, R)}{\mathcal{D}, R, G \vdash \{p\}\langle C \rangle\{q\}} \ (\textsc{atom-r})$$

Figure 7. Inference rules.

requires the thread be waiting for its environment to perform a finite number of definite actions.

Definition 1 (Definite Progress). $\mathfrak{S} \models (R, G : \mathcal{D} \xrightarrow{f} Q)$ iff the following hold for all t:

(1) either $\mathfrak{S} \models Q_{\mathsf{t}}$,
or there exists t' such that $\mathsf{t}' \neq \mathsf{t}$ and $\mathfrak{S} \models \mathsf{Enabled}(\mathcal{D}_{\mathsf{t}'})$;
(2) for any $\mathsf{t}' \neq \mathsf{t}$ and $\mathfrak{S}'$, if $(\mathfrak{S}, \mathfrak{S}', 0) \models R_{\mathsf{t}} \wedge \langle \mathcal{D}_{\mathsf{t}'} \rangle$, then $f_{\mathsf{t}}(\mathfrak{S}') < f_{\mathsf{t}}(\mathfrak{S})$;
(3) for any $\mathfrak{S}'$, if $(\mathfrak{S}, \mathfrak{S}', 0) \models R_{\mathsf{t}} \vee G_{\mathsf{t}}$, then $f_{\mathsf{t}}(\mathfrak{S}') \leq f_{\mathsf{t}}(\mathfrak{S})$.

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

Ignoring the index 0 above, the definition says either Q holds over $\mathfrak{S}$ or the definite action $\mathcal{D}_{\mathsf{t}'}$ of some environment thread t' is enabled. Also we require the metric $f(\mathfrak{S})$ to decrease when a definite action is performed. Besides, the metric should never increase at any step of the execution.

To ensure that the metric f decreases regardless of the time when the environment's definite actions take place, the definite progress should always hold. This is enforced by finding a stronger assertion J such that $p \Rightarrow J$ and $\mathsf{Sta}(J, R \vee G)$ hold, that is, J is an invariant that holds at every program point of the loop. If $(R, G : \mathcal{D} \xrightarrow{f} Q)$ happens to satisfy the two premises, we can use $(R, G : \mathcal{D} \xrightarrow{f} Q)$ directly as J, but in practice it could be easier to prove the stability requirement by finding a stronger J. The definition of stability $\mathsf{Sta}(p, R)$ is given in Fig. 8.

Notice in $(R, G : \mathcal{D}' \xrightarrow{f} Q)$ we can use $\mathcal{D}'$ instead of $\mathcal{D}$ to simplify the proof, as long as $\mathcal{D}' \leqslant \mathcal{D}$ and $\mathsf{wffAct}(R, \mathcal{D}')$ are satisfied. The premise $\mathcal{D}' \leqslant \mathcal{D}$, defined in Fig. 6, says $\mathcal{D}'$ specifies a subset of definite actions in $\mathcal{D}$. For instance, if $\mathcal{D}$ consists of multiple definite actions and is in the form of $\mathcal{D}_1 \wedge \ldots \wedge \mathcal{D}_n$, $\mathcal{D}'$ may contain only a subset of these $\mathcal{D}_k$ $(1 \leq k \leq n)$. The way to exclude in $\mathcal{D}'$ irrelevant definite actions can simplify the proof of the condition (2) of definite progress (see Definition 1).

Given the definite progress condition, we know Q will be eventually true because each definite action will definitely happen. Then the loop starts to consume $\Diamond$-tokens and needs to finally terminate, following our argument at the beginning.

4.2.3 More Inference Rules

The $\textsc{hide-}\Diamond$ rule allows us to discard the tokens (by using $\lfloor \_ \rfloor_{\Diamond}$) when the termination of code C is already established, which is useful for modular verification of nested loops.

ATOM *rules for refinement reasoning.* The ATOM rule allows us to logically execute the abstract code simultaneously with every concrete step (let's first ignore the index k in the premises of the rule). We use $\vdash [p]C[q]$ to represent the total correctness of C in

$\mathsf{wffAct}(R, \mathcal{D})$ iff $\forall \mathsf{t}.\, R_{\mathsf{t}} \Rightarrow [\mathcal{D}_{\mathsf{t}}] \wedge (\forall \mathsf{t}' \neq \mathsf{t}.\, [\mathcal{D}_{\mathsf{t}'}] \vee \mathcal{D}_{\mathsf{t}'})$

$\mathsf{Sta}(p, R)$ iff $\forall \mathfrak{S}, \mathfrak{S}', u, w, C.$
$\quad ((\mathfrak{S}, (u, w), C) \models p) \wedge ((\mathfrak{S}, \mathfrak{S}') \models R) \Longrightarrow (\mathfrak{S}', (u, w), C) \models p$

$p \Rrightarrow q$ iff $\forall \mathsf{t}, \sigma, \Sigma, u, w, C, \Sigma_F.$
$\quad (((\sigma, \Sigma), (u, w), C) \models p) \wedge (\Sigma \bot \Sigma_F) \Longrightarrow \exists C', \Sigma'.$
$\quad\quad ((C, \Sigma \uplus \Sigma_F) \longrightarrow\!\!\triangleright^*_{\mathsf{t}} (C', \Sigma' \uplus \Sigma_F)) \wedge ((\sigma, \Sigma'), (u, w), C') \models q$

Figure 8. Auxiliary defs. used in logic rules (simplified version).

sequential separation logic. The corresponding rules are standard and elided here. We use $p \Rrightarrow q$ for the zero or multiple-step executions from the abstract code specified by p to the code specified by q, which is defined in Fig. 8. Then, the ATOM rule allows us to execute zero-or-more steps of the abstract code with the execution of C, as long as the overall transition (including the abstract steps and the concrete steps) satisfies the relational guarantee G. We can lift C's total correctness to the concurrent setting as long as the environment consists of identity transitions only. To allow a weaker R, we can apply the ATOM-R rule later, which requires that the pre- and post-conditions be stable with respect to R.

4.2.4 Example: Ticket Locks

We prove the starvation-freedom of the ticket lock implementation in Fig. 9 using our logic rules. We have informally discussed in Sec. 2 the verification of the counter using a ticket lock (`sfInc` in Fig. 1(c)). To simplify the presentation, here we erase the increment in the critical section and focus on the progress property of the code in Fig. 9. With an empty critical section, the code functions just as **skip**, so Fig. 9 proves it is linearizable with respect to **skip**. The proofs of `sfInc` (including its starvation-freedom and linearizability with respect to the atomic `INC` in Fig. 1(a)) are given in TR [22].

To help specify the queue of the threads requesting the lock, we introduce an auxiliary array `ticket`. As shown in Fig. 9, each array cell `ticket`$[i]$ records the ID of the unique thread which gets the ticket number i. Here we use `cid` for the current thread ID.

We then define some predicates to describe the lock status. $\mathsf{lock}(\mathit{tl}, n_1, n_2)$ contains the auxiliary `ticket` array in addition to `owner` and `next`, where `owner` $\mapsto n_1$ and `next` $\mapsto n_2$, and tl is the list of the threads recorded in `ticket`$[n_1], \ldots,$ `ticket`$[n_2 - 1]$. We also use $\mathsf{locked}(\mathit{tl}, n_1, n_2)$ for the case when tl is not empty. That is, the lock is acquired by the first thread in tl, while the other threads in tl are waiting for the lock in order. Besides, we use $\mathsf{lockIrr}_{\mathsf{t}}(\mathit{tl}, n_1, n_2)$ short for $\mathsf{lock}(\mathit{tl}, n_1, n_2) \wedge (\mathsf{t} \notin \mathit{tl})$. That is, the thread t is "irrelevant" to the lock: it does not request the lock. The formal definitions are given in TR [22].

The bottom of Fig. 9 defines the precondition P and the guarantee condition G of the code. G_{t} specifies the possible atomic actions of a thread t. $\mathit{Lock}_{\mathsf{t}}$ adds the thread t at the end of tl of the threads requesting the lock and increments `next`. It corresponds to line 2

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

$$P_{\mathsf{t}} \overset{\text{def}}{=} \exists tl, n_1, n_2.\, \mathsf{lockIrr_t}(tl, n_1, n_2) \qquad G_{\mathsf{t}} \overset{\text{def}}{=} \mathit{Lock}_{\mathsf{t}} \vee \mathit{Unlock}_{\mathsf{t}}$$

$$\mathit{Lock}_{\mathsf{t}} \overset{\text{def}}{=} \exists tl, n_1, n_2.\, \mathsf{lockIrr_t}(tl, n_1, n_2) \ltimes \mathsf{locked}(tl :: \mathsf{t}, n_1, n_2 + 1)$$

$$\mathit{Unlock}_{\mathsf{t}} \overset{\text{def}}{=} \exists tl, n_1, n_2.\, \mathsf{locked}(\mathsf{t} :: tl, n_1, n_2) \ltimes \mathsf{lockIrr_t}(tl, n_1{+}1, n_2)$$

$$\mathcal{D}_{\mathsf{t}} \overset{\text{def}}{=} \forall tl, n_1, n_2.\, \mathsf{locked}(\mathsf{t} :: tl, n_1, n_2) \rightsquigarrow \mathsf{lockIrr_t}(tl, n_1 + 1, n_2)$$

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

$$Q_{\mathsf{t}} \overset{\text{def}}{=} \exists n_2, tl_2.\, \mathsf{locked}(\mathsf{t} :: tl_2, \mathtt{i}, n_2) \wedge (\mathtt{o} \le \mathtt{i})$$

$$f(\mathfrak{S}) = k \text{ iff } \mathfrak{S} \models \exists n_1.\, (\mathtt{owner} \mapsto n_1) * \mathsf{true} \wedge (\mathtt{i} - n_1 = k)$$

Figure 9. Proofs for the ticket lock (with auxiliary code in gray).

of Fig. 9. $\mathit{Unlock}_{\mathsf{t}}$ releases the lock by incrementing `owner` and dequeuing the thread t which currently holds the lock. It corresponds to line 4 of Fig. 9. The rely condition R_{t} includes all the $G_{\mathsf{t}'}$ made by the environment threads t'.

Next we define the definite action $\mathcal{D}$. As we explained in Sec. 2, $\mathcal{D}_{\mathsf{t}}$ requires that whenever the thread t holds a lock with $\mathtt{owner} \mapsto n_1$, it should eventually release the lock by incrementing `owner` to $n_1 + 1$. We can prove the side conditions about well-formedness of specifications in the OBJ rule hold.

The key to verifying the loop at line 3 is to find a metric function f and prove definite progress $J \Rightarrow (R, G' : \mathcal{D} \xrightarrow{f} Q)$ for a stable J. As shown in Fig. 9, we define J_{t} to say that the thread t is requesting the lock. Here $\mathsf{tlocked}_{tl_1, \mathsf{t}, tl_2}(n_1, \mathtt{i}, n_2)$ is similar to $\mathsf{locked}(tl_1 :: \mathsf{t} :: tl_2, n_1, n_2)$. It also says that the thread t takes the ticket number `i`. Q_{t} specifies the case when tl_1 is empty (thus $\mathtt{owner} \mapsto \mathtt{i}$). We also strengthen the guarantee condition G' of the loop to Id, the identity transitions.

The metric function f maps each state $\mathfrak{S}$ to the difference between `i` and `owner` at that state, which describes the number of threads ahead of t in the waiting queue. We use the usual $<$ order on natural numbers as the associated well-founded order. Then, we can verify $J \Rightarrow (R, G' : \mathcal{D} \xrightarrow{f} Q)$.

Finally, we prove that the loop terminates with one $\Diamond$-token when Q holds or $\mathcal{D}$ is enabled. Then we can conclude linearizability and starvation-freedom of the ticket lock implementation in Fig. 9.

4.3 Adding Delay for Deadlock-Free Objects

As we explained in Sec. 2, deadlock-free objects allow the progress of a thread to be delayed by its environment, as long as the whole system makes progress. Correspondingly, to verify deadlock-free objects, we extend our logic with a delay mechanism. First we find out the delaying actions and stratify them for objects with rollbacks where a delaying action may trigger more steps of other delaying actions. Then, we introduce $\blacklozenge$-tokens (we use $\blacklozenge$ here to distinguish them from $\Diamond$-tokens for loops) to bound the number of delaying actions in each method, so we avoid infinite delays without whole-system progress.

Multi-level rely/guarantee assertions. As shown in Fig. 4, we index the rely/guarantee assertion $P \ltimes_k Q$ with a natural number k and call it a level-k action. We require $0 \le k < \mathsf{maxL}$, where maxL is a predefined upper bound of all levels. Intuitively, $P \ltimes_k Q$ could make other threads do more actions at a level $k' < k$. Thus $P \ltimes_0 Q$ cannot make other threads do any more actions, i.e., it cannot delay other threads. $P \ltimes_1 Q$ could make other threads do more actions at level 0 but no more at level 1, thus we avoid the problem of delay-caused circular dependency discussed in Sec. 2.2.2.

$$\mathcal{L}((\mathfrak{S}, \mathfrak{S}'), P \ltimes_k Q) \overset{\text{def}}{=} \begin{cases} k & \text{if } (\mathfrak{S}, \mathfrak{S}') \models P \ltimes_k Q \\ \mathsf{maxL} & \text{otherwise} \end{cases}$$

$$\mathcal{L}((\mathfrak{S}, \mathfrak{S}'), [P]) \overset{\text{def}}{=} \begin{cases} 0 & \text{if } (\mathfrak{S}, \mathfrak{S}') \models [P] \\ \mathsf{maxL} & \text{otherwise} \end{cases}$$

$$\mathcal{L}((\mathfrak{S}, \mathfrak{S}'), \mathcal{D}) \overset{\text{def}}{=} \begin{cases} 0 & \text{if } (\mathfrak{S}, \mathfrak{S}') \models \mathcal{D} \\ \mathsf{maxL} & \text{otherwise} \end{cases}$$

$$\mathcal{L}((\mathfrak{S}, \mathfrak{S}'), R \wedge R') \overset{\text{def}}{=} \mathsf{max}(\mathcal{L}((\mathfrak{S}, \mathfrak{S}'), R), \mathcal{L}((\mathfrak{S}, \mathfrak{S}'), R'))$$

$$\mathcal{L}((\mathfrak{S}, \mathfrak{S}'), R \vee R') \overset{\text{def}}{=} \mathsf{min}(\mathcal{L}((\mathfrak{S}, \mathfrak{S}'), R), \mathcal{L}((\mathfrak{S}, \mathfrak{S}'), R'))$$

$$\mathcal{L}((\mathfrak{S}, \mathfrak{S}'), \lfloor R \rfloor_0) \overset{\text{def}}{=} \begin{cases} 0 & \text{if } \mathcal{L}((\mathfrak{S}, \mathfrak{S}'), R) = 0 \\ \mathsf{maxL} & \text{otherwise} \end{cases}$$

$$(\mathfrak{S}, \mathfrak{S}') \models \lfloor R \rfloor_0 \ \text{ iff } \ \mathcal{L}((\mathfrak{S}, \mathfrak{S}'), R) = 0$$

$$(\mathfrak{S}, \mathfrak{S}', k) \models R \ \text{ iff } \ \mathcal{L}((\mathfrak{S}, \mathfrak{S}'), R) = k \text{ and } k < \mathsf{maxL}$$

$$R \Rightarrow R' \qquad \text{iff } \forall \mathfrak{S}, \mathfrak{S}', k.\, ((\mathfrak{S}, \mathfrak{S}', k) \models R) \Longrightarrow (\mathfrak{S}, \mathfrak{S}', k) \models R'$$

$$u ::= (n_k, \ldots, n_1) \quad (1 \le k < \mathsf{maxL})$$

$$(n'_m, \ldots, n'_1) <_k (n_m, \ldots, n_1) \text{ iff } (\forall i > k.\, (n'_i = n_i)) \wedge (n'_k < n_k)$$

$$(n'_m, \ldots, n'_1) \approx_k (n_m, \ldots, n_1) \text{ iff } (\forall i \ge k.\, (n'_i = n_i))$$

$$u < u' \text{ iff } \exists k.\, u <_k u' \qquad u \le u' \text{ iff } u < u' \vee u = u'$$

$$(u, w) <_k (u', w') \text{ iff } (u <_k u') \vee (k = 0 \wedge u = u' \wedge w = w')$$

$$(u, w) \approx_k (u', w') \text{ iff } u \approx_k u' \wedge (k = 0 \Longrightarrow w = w')$$

Figure 10. Levels of state transitions and tokens.

To interpret the level numbers in the assertion semantics, we define $\mathcal{L}((\mathfrak{S}, \mathfrak{S}'), R)$ in Fig. 10 which assigns a level to the transition $(\mathfrak{S}, \mathfrak{S}')$, given the specification R. That is, if $\mathcal{L}((\mathfrak{S}, \mathfrak{S}'), R) = k$, we say R views $(\mathfrak{S}, \mathfrak{S}')$ as a level-k transition. We let $k = \mathsf{maxL}$ if the transition does not satisfy R. Given the level function, we can now define the semantics of $\lfloor R \rfloor_0$, which picks out the transitions that R views as level-0 ones. For the following example R,

$$R \overset{\text{def}}{=} (P \ltimes_0 Q) \vee (P' \ltimes_1 Q')$$

$\lfloor R \rfloor_0$ is equivalent to $P \ltimes_0 Q$. Besides, $R \Rightarrow \lfloor R \rfloor_0$ means that R views all state transitions as level-0 ones, thus any state transitions of R should not delay the progress of other threads.

We use $(\mathfrak{S}, \mathfrak{S}', k) \models R$ as a shorthand for $\mathcal{L}((\mathfrak{S}, \mathfrak{S}'), R) = k$ ($k < \mathsf{maxL}$). Then the implication $R \Rightarrow R'$ is redefined under this new interpretation, as shown in Fig. 10.

$\blacklozenge$-tokens in assertions. To ensure the progress of the whole system, we require the steps of delaying actions to pay $\blacklozenge$-tokens. Since we allow multi-levels of transitions to delay other threads, the $\blacklozenge$-tokens are stratified accordingly. Thus we introduce the new assertion $\blacklozenge(E_k, \ldots, E_1)$ in Fig. 4, whose semantics is defined in Fig. 5. It says the number of each level-j $\blacklozenge$-tokens is *no less than* E_j. Here u is a sequence $(n_k, \ldots, n_1)$ recording the numbers of $\blacklozenge$-tokens at different levels, as defined in Fig. 10. We overload $<$ as the dictionary order for the sequence of natural numbers. The ordering over u and other related definitions are also given in Fig. 10.

4.3.1 Inference Rules Revisited

To use the logic to verify deadlock-free objects, we need to first find in each object method the actions that may delay the progress of others. Since some of these actions may be further delayed by others, we assign levels to them to ensure each action can only be delayed by ones at higher levels. We specify the actions and their levels in the rely/guarantee conditions. We also need to decide an upper bound of these execution steps at each level and specify them as the number of $\blacklozenge$-tokens in the precondition of each method.

Below we revisit the inference rules in Fig. 7 and explain their use of multi-level actions and $\blacklozenge$-tokens. In the OBJ rule, we specify in the

$$\mathsf{wffAct}(R, \mathcal{D}) \ \text{ iff } \ \forall \mathsf{t}.\, \lfloor R_{\mathsf{t}} \rfloor_0 \Rightarrow [\mathcal{D}_{\mathsf{t}}] \wedge (\forall \mathsf{t}' \neq \mathsf{t}.\, [\mathcal{D}_{\mathsf{t}'}] \vee \mathcal{D}_{\mathsf{t}'})$$

$$\begin{aligned}
&p \Rrightarrow_k q \ \text{ iff } \ \forall \mathsf{t}, \sigma, \Sigma, u, w, C, \Sigma_F. \\
&\qquad (((\sigma, \Sigma), (u, w), C) \models p) \wedge (\Sigma \bot \Sigma_F) \implies \exists u', w', C', \Sigma'. \\
&\qquad ((C, \Sigma \uplus \Sigma_F) \longrightarrow^*_{\mathsf{t}} (C', \Sigma' \uplus \Sigma_F)) \wedge (((\sigma, \Sigma'), (u', w'), C') \models q) \\
&\qquad \wedge\, (u', w') <_k (u, w) \qquad\quad (<_k \text{ defined in Fig. 10})
\end{aligned}$$

$$\begin{aligned}
&\mathsf{Sta}(p, R) \ \text{ iff } \ \forall \mathfrak{S}, \mathfrak{S}', u, w, C, k. \\
&\qquad ((\mathfrak{S}, (u, w), C) \models p) \wedge ((\mathfrak{S}, \mathfrak{S}', k) \models R) \implies \exists u', w'. \\
&\qquad ((\mathfrak{S}', (u', w'), C) \models p) \ \wedge\ ((u', w') \approx_k (u, w)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{where } \approx_k \text{ is defined in Fig. 10.}
\end{aligned}$$

Figure 11. Key auxiliary definitions for inference rules (final version that supersedes definitions in Fig. 8).

precondition the number of $\blacklozenge$-tokens needed for each object method. The side condition $\mathsf{wffAct}(R, \mathcal{D})$ is also redefined in Fig. 11, which is explained below.

Decreasing $\blacklozenge$-tokens at the* ATOM *rule. The thread loses $\blacklozenge$-tokens when it performs an action that may delay other threads. This is required by the ATOM rule. Depending on whether the atomic command may delay others or not, we assign a level k in the premise $q' \Rrightarrow_k q$, which is redefined in Fig. 11. Similar to $p \Rrightarrow q$ in Fig. 8, it allows us to execute the abstract code. Now it also requires the number of $\blacklozenge$-tokens at level k to be decreased if $k \geq 1$.

Note k cannot be arbitrarily chosen. The assignment of the level k to the atomic operation must be consistent with the level specification in G, as required by the third premise.

Being delayed: increasing tokens at stability checking. When the progress of the thread t is delayed by a level-k ($k \geq 1$) action from its environment thread t', thread t could gain more $\lozenge$-tokens to do more loop iterations. It could also gain more level-k' ($k' < k$) $\blacklozenge$-tokens to execute more level-k' actions. Here increasing tokens would not affect the soundness of our logic because the environment thread t' must pay a higher-level token for its higher-level delaying action. As we explained in Sec. 2.3.4, the $\blacklozenge$-tokens at all levels in the whole system actually form a tuple which strictly descends along the dictionary order, ensuring the whole-system progress.

We re-define the stability $\mathsf{Sta}(p, R)$ in Fig. 11 to reflect the possible increasing of tokens for the thread t. We could reset w and the number at level $k' < k$ in u after the environment step R if this step is associated with a level k ($k \geq 1$).

Allowing queue jumps at definite progress and wffAct. As we explained in Sec. 2.3.3, for deadlock-free objects, the environment steps could cause queue jumps to delay the progress of the thread t. Like starvation-free objects, the thread t using deadlock-free objects may wait for a queue of definite actions made by its environment. A queue jump would make the thread t wait for a longer queue of the environment's definite actions.

As shown in Definition 1, the definite progress condition ($R, G\colon \mathcal{D} \xrightarrow{f} Q$) allows the thread to reset its metric $f(\mathfrak{S})$ for a queue jump when the environment step is associated with level $k \geq 1$ (i.e., it is a delaying action). In this case, although the current thread may be blocked for a longer time, the whole system must progress since a $\blacklozenge$-token is paid by the environment thread for the delaying action.

Also the requirement $\mathsf{wffAct}(R, \mathcal{D})$ (used at the OBJ rule and the WHL rule) should be revised to allow queue jumps. The new definition is shown in Fig. 11. Here we allow a queue jump to disable the definite action $\mathcal{D}$ of the thread at the head of the queue, so it is not necessary to require $\mathsf{Enabled}(\mathcal{D})$ to be preserved when the environment step is associated with level $k \geq 1$.

4.3.2 Example: Test-and-Set Locks

In Fig. 12, we verify the test-and-set lock implementation explained in Sec. 2. Like the ticket lock proofs in Sec. 4.2.4, we simplify the

$$\begin{array}{ll}
\mathsf{locked_t} \overset{\text{def}}{=} (\mathtt{L} \mapsto \mathsf{t}) & \mathsf{envLocked_t} \overset{\text{def}}{=} \exists \mathsf{t}'.\, \mathsf{locked}_{\mathsf{t}'} \wedge (\mathsf{t}' \neq \mathsf{t}) \\
\mathsf{unlocked} \overset{\text{def}}{=} (\mathtt{L} \mapsto 0) & \mathsf{notOwn_t} \overset{\text{def}}{=} \mathsf{unlocked} \vee \mathsf{envLocked_t}
\end{array}$$

$$G_{\mathsf{t}} \overset{\text{def}}{=} \mathit{Lock}_{\mathsf{t}} \vee \mathit{Unlock}_{\mathsf{t}}$$

$$\begin{array}{ll}
\mathit{Lock}_{\mathsf{t}} \overset{\text{def}}{=} \mathsf{unlocked} \ltimes_1 \mathsf{locked_t} & \mathit{Unlock}_{\mathsf{t}} \overset{\text{def}}{=} \mathsf{locked_t} \ltimes_0 \mathsf{unlocked} \\
\mathcal{D}_{\mathsf{t}} \overset{\text{def}}{=} \mathsf{locked_t} \leadsto \mathsf{unlocked} & J_{\mathsf{t}} \overset{\text{def}}{=} \mathsf{notOwn_t} \vee \mathsf{locked_t} \\
Q_{\mathsf{t}} \overset{\text{def}}{=} \mathsf{unlocked} \vee \mathsf{locked_t} & f_{\mathsf{t}}(\mathfrak{S}) = \begin{cases} 1 & \text{if } \mathfrak{S} \models \mathsf{envLocked_t} \\ 0 & \text{if } \mathfrak{S} \models Q_{\mathsf{t}} \end{cases}
\end{array}$$

```
   $\{ \mathsf{notOwn_{cid}} \wedge \blacklozenge \}$
1  local b := false;
   $\{ ((\neg \mathsf{b}) \wedge \mathsf{notOwn_{cid}} \wedge \blacklozenge \wedge \lozenge) \vee (\mathsf{b} \wedge \mathsf{locked_{cid}}) \}$
2  while (!b) {
     $\{ (\mathsf{unlocked} \wedge \blacklozenge) \vee (\mathsf{envLocked_{cid}} \wedge \blacklozenge \wedge \lozenge) \}$
3    b := cas(L, 0, cid);
     $\{ (\mathsf{b} \wedge \mathsf{locked_{cid}}) \vee ((\neg \mathsf{b}) \wedge \mathsf{notOwn_{cid}} \wedge \blacklozenge \wedge \lozenge) \}$
4  }
   $\{ \mathsf{locked_{cid}} \}$
5  [L] := 0;
   $\{ \mathsf{notOwn_{cid}} \}$
```

Figure 12. Proofs for the TAS lock.

code and prove it is linearizable with respect to **skip**. Here we omit the assertion $\mathsf{arem}(\textbf{skip})$ at each line in the proof, and focus on proving deadlock-freedom of the code.

As defined at the top of Fig. 12, the action $\mathsf{Lock_t}$ (corresponding to the successful cas at line 3) is at level 1, which may delay other threads trying to acquire the lock. The $\mathsf{Unlock_t}$ action is at level 0, which cannot delay other threads. Also the precondition is given a $\blacklozenge$-token, which is required to pay for the $\mathsf{Lock_t}$ action.

The definite action $\mathcal{D}$ simply says that the thread t would eventually release the lock when it acquires the lock. It is easy to check that the side conditions about R, G and $\mathcal{D}$ in the OBJ rule, e.g., $\mathsf{wffAct}(R, \mathcal{D})$, are satisfied.

$R, G\colon \mathcal{D} \xrightarrow{f} Q$ specifies the queue of definite actions which now contains at most one environment thread. That is, the metric $f(\mathfrak{S})$ is 1 if the lock is not available, and is 0 otherwise. When an environment thread t' cuts in line by acquiring the lock when the lock is free, the current thread t has to wait for $\mathcal{D}_{\mathsf{t}'}$ before t itself progresses. Thus in $R, G\colon \mathcal{D} \xrightarrow{f} Q$ the current thread t can reset its metric $f(\mathfrak{S})$ when its environment acquires the lock.

The detailed proof at the bottom of Fig. 12 shows the changes of tokens. We give the current thread one $\lozenge$-token (using the HIDE-$\lozenge$ rule) to do its loop at lines 2-4. It consumes this $\lozenge$-token at the beginning of the loop body when Q holds, as shown in the left branch of the assertion p before line 3. When Q does not hold, as shown in p's right branch, the loop does not consume the $\lozenge$-token.

Next we stabilize p. For the left branch, if an environment thread t' acquires the lock, which is a delaying action $\mathit{Lock}_{\mathsf{t}'}$, we let the current thread *regain* a $\lozenge$-token. The resulting state just satisfies the right branch of p. Thus p is already stable.

The current thread pays its $\blacklozenge$-token when its cas at line 3 succeeds (i.e., it acquires the lock), as shown in the left branch of the assertion after line 3. If the cas fails, the thread still has $\blacklozenge$ to acquire the lock in the future and $\lozenge$ to try one more iteration.

4.3.3 Another Example: Nested Locks with Rollback

To demonstrate the use of action levels, we verify the rollback code in Fig. 2(b) which we informally discussed in Sec. 2. Here we assume all the methods of the object either acquire L1 before L2 (as in the method of Fig. 2(b)), or acquire only one lock.

Stratified delaying actions. As in the TAS lock example in Sec. 4.3.2, lock acquirements are delaying actions. Here we have

$$\begin{aligned}
G_{\mathtt{t}} &\overset{\text{def}}{=} \mathit{Lock2}_{\mathtt{t}} \vee \mathit{Lock1}_{\mathtt{t}} \vee \mathit{Lock0}_{\mathtt{t}} \vee \mathit{Unlock2}_{\mathtt{t}} \vee \mathit{Unlock1}_{\mathtt{t}}\\
\mathit{Lock2}_{\mathtt{t}} &\overset{\text{def}}{=} (\mathsf{unlocked}(\mathtt{L2}) \ltimes_2 \mathsf{locked}_{\mathtt{t}}(\mathtt{L2})) * [\mathtt{L1} \mapsto \_]\\
\mathit{Lock1}_{\mathtt{t}} &\overset{\text{def}}{=} (\mathsf{unlocked}(\mathtt{L1}) \ltimes_1 \mathsf{locked}_{\mathtt{t}}(\mathtt{L1})) * [\mathsf{unlocked}(\mathtt{L2})]\\
\mathit{Lock0}_{\mathtt{t}} &\overset{\text{def}}{=} (\mathsf{unlocked}(\mathtt{L1}) \ltimes_0 \mathsf{locked}_{\mathtt{t}}(\mathtt{L1})) * [\mathsf{envLocked}_{\mathtt{t}}(\mathtt{L2})]\\
\mathit{Unlock2}_{\mathtt{t}} &\overset{\text{def}}{=} (\mathsf{locked}_{\mathtt{t}}(\mathtt{L2}) \ltimes_0 \mathsf{unlocked}(\mathtt{L2})) * [\mathtt{L1} \mapsto \_]\\
\mathit{Unlock1}_{\mathtt{t}} &\overset{\text{def}}{=} (\mathsf{locked}_{\mathtt{t}}(\mathtt{L1}) \ltimes_0 \mathsf{unlocked}(\mathtt{L1})) * [\mathtt{L2} \mapsto \_]\\
\mathcal{D}_{\mathtt{t}} &\overset{\text{def}}{=} \mathcal{D}2_{\mathtt{t}} \wedge \mathcal{D}1_{\mathtt{t}}\\
\mathcal{D}2_{\mathtt{t}} &\overset{\text{def}}{=} \forall s.\ \mathsf{locked}_{\mathtt{t}}(\mathtt{L2}) * (\mathtt{L1} \mapsto s) \leadsto \mathsf{unlocked}(\mathtt{L2}) * (\mathtt{L1} \mapsto s)\\
\mathcal{D}1_{\mathtt{t}} &\overset{\text{def}}{=} \mathsf{locked}_{\mathtt{t}}(\mathtt{L1}) * \mathsf{unlocked}(\mathtt{L2}) \leadsto \mathsf{unlocked}(\mathtt{L1}) * \mathsf{unlocked}(\mathtt{L2})
\end{aligned}$$

Figure 13. Multi-level actions for the example in Fig. 2(b).

```
     $\{\,\mathsf{notOwn_{cid}}(\mathtt{L1}) * \mathsf{notOwn_{cid}}(\mathtt{L2}) \wedge \blacklozenge(1,1)\,\}$
1    lock L1;
     $p_1 \overset{\text{def}}{=} \left\{ \begin{array}{l} \mathsf{locked_{cid}}(\mathtt{L1}) * (\mathsf{unlocked}(\mathtt{L2}) \wedge \blacklozenge(1,0) \\ \qquad\qquad \vee\, \mathsf{envLocked_{cid}}(\mathtt{L2}) \wedge \blacklozenge(1,1)) \end{array} \right\}$
2    local r := L2;
     $\left\{ \begin{array}{l} \mathsf{locked_{cid}}(\mathtt{L1}) * \mathsf{notOwn_{cid}}(\mathtt{L2}) \\ \wedge\, ((\mathtt{r}=0) \wedge \blacklozenge(1,0) \ \vee\ (\mathtt{r} \neq 0) \wedge \blacklozenge(1,1) \wedge \lozenge) \end{array} \right\}$
3    while (r != 0) {
       $p_2 \overset{\text{def}}{=} \left\{ \begin{array}{l} \mathsf{locked_{cid}}(\mathtt{L1}) \\ * (\mathsf{unlocked}(\mathtt{L2}) \vee \mathsf{envLocked_{cid}}(\mathtt{L2}) \wedge \lozenge) \wedge \blacklozenge(1,1) \end{array} \right\}$
4      unlock L1;
5      lock L1;
       $\left\{ \begin{array}{l} \mathsf{locked_{cid}}(\mathtt{L1}) * (\mathsf{unlocked}(\mathtt{L2}) \wedge \blacklozenge(1,0) \\ \qquad\qquad \vee\, \mathsf{envLocked_{cid}}(\mathtt{L2}) \wedge \blacklozenge(1,1) \wedge \lozenge) \end{array} \right\}$
6      r := L2;
       $\left\{ \begin{array}{l} \mathsf{locked_{cid}}(\mathtt{L1}) * \mathsf{notOwn_{cid}}(\mathtt{L2}) \\ \wedge\, ((\mathtt{r}=0) \wedge \blacklozenge(1,0) \ \vee\ (\mathtt{r}=1) \wedge \blacklozenge(1,1) \wedge \lozenge) \end{array} \right\}$
7    }
     $\{\,\mathsf{locked_{cid}}(\mathtt{L1}) * \mathsf{notOwn_{cid}}(\mathtt{L2}) \wedge \blacklozenge(1,0)\,\}$
8    lock L2;
     $\{\,\mathsf{locked_{cid}}(\mathtt{L1}) * \mathsf{locked_{cid}}(\mathtt{L2})\,\}$
9    unlock L2;
10   unlock L1;
     $\{\,\mathsf{notOwn_{cid}}(\mathtt{L1}) * \mathsf{notOwn_{cid}}(\mathtt{L2})\,\}$

$Q_{\mathtt{t}} \overset{\text{def}}{=} (\mathsf{locked_t}(\mathtt{L1}) \vee \mathsf{unlocked}(\mathtt{L1})) * \mathsf{unlocked}(\mathtt{L2})$
```

Figure 14. Proof outline for the rollback example in Fig. 2(b).

two locks L1 and L2, and a thread may roll back and re-acquire L1 if its environment owns L2. To support the rollbacks, we stratify the delaying actions and $\blacklozenge$-tokens in two levels. Acquirements of L2 are at level 2, defined as *Lock2* in Fig. 13, which may trigger rollbacks and more acquirements of L1. Acquirements of L1 at level 1 may delay other threads requesting L1, causing them to do more non-delaying actions, but cannot reversely trigger more level-2 actions. Thus we avoid the circular delay problem.

However, acquirements of L1 in some special cases cannot be viewed as delaying actions. Suppose L2 is acquired by an environment thread t′ before the current thread t starts the method. Then t would continuously roll back until t′ releases L2. It may acquire L1 infinitely often. In this case, viewing all acquirements of L1 as delaying actions would require t to pay $\blacklozenge$-tokens infinitely often, and consequently require an infinite number of $\blacklozenge$-tokens be assigned to the method at the beginning, which is impossible. To address the problem, we define in Fig. 13 that acquiring L1 is a level-1 action *Lock1 only if* L2 is free. When L2 is acquired by the environment, we say the current thread is "blocked", and we view its acquirement of L1 as a non-delaying action *Lock0* at level 0.

To simplify the presentation, the definitions in Fig. 13 follow the notations in LRG [7], using "$* \, [P]$" to mean that the actions on the irrelevant part P of the states are identity transitions.

```
$p_{01} \overset{\text{def}}{=} \mathsf{unlocked}(\mathtt{L1}) \qquad\quad * \mathsf{unlocked}(\mathtt{L2})$
$p_{02} \overset{\text{def}}{=} (\mathsf{unlocked}(\mathtt{L1}) \qquad * \mathsf{envLocked_{cid}}(\mathtt{L2})) \wedge \lozenge$
$p_{03} \overset{\text{def}}{=} (\mathsf{envLocked_{cid}}(\mathtt{L1}) * \mathsf{notOwn_{cid}}(\mathtt{L2})) \quad \wedge \lozenge$
    $\{\,\mathsf{notOwn_{cid}}(\mathtt{L1}) * \mathsf{notOwn_{cid}}(\mathtt{L2}) \wedge \blacklozenge(1,1)\,\}$
1   local b := false;
    $\left\{ \begin{array}{l} (\neg\mathtt{b}) \wedge (\mathsf{notOwn_{cid}}(\mathtt{L1}) * \mathsf{notOwn_{cid}}(\mathtt{L2})) \wedge \blacklozenge(1,1) \wedge \lozenge \\ \vee\, \mathtt{b} \wedge p_1 \end{array} \right\}$
2   while (!b) {
      $\{\,(p_{01} \vee p_{02} \vee p_{03}) \wedge \blacklozenge(1,1)\,\}$
3     b := cas(L1, 0, cid);
      $\left\{ \begin{array}{l} \mathtt{b} \wedge \mathsf{locked_{cid}}(\mathtt{L1}) \\ * (\mathsf{unlocked}(\mathtt{L2}) \wedge \blacklozenge(1,0) \vee \mathsf{envLocked_{cid}}(\mathtt{L2}) \wedge \blacklozenge(1,1)) \\ \vee\, (\neg\mathtt{b}) \wedge ((\mathsf{unlocked}(\mathtt{L1}) \vee \mathsf{envLocked_{cid}}(\mathtt{L1})) \\ \qquad\qquad * \mathsf{notOwn_{cid}}(\mathtt{L2})) \wedge \blacklozenge(1,1) \wedge \lozenge \end{array} \right\}$
4   }
    $\{\,p_1\,\}$
```

Figure 15. Proof outline for `lock L1` in the rollback example.

Definite actions. There are two kinds of definite actions, which release the two locks respectively. As shown in Fig. 13, $\mathcal{D}2$ says a thread holding L2 eventually releases it, regardless of the status of the lock L1. $\mathcal{D}1$ says L1 will be definitely released when L2 is free. Note that a thread holding L1 may not be able to release the lock if it cannot acquire L2.

Proof outline for the rollback. As shown in Fig. 14, when thread t starts the method, it is given $\blacklozenge(1,1)$, where the level-2 $\blacklozenge$-token is for doing *Lock2* and the level-1 $\blacklozenge$-token is for *Lock1*. The assertions notOwn is defined similarly as in Fig. 12.

`lock L1` at line 1 is implemented using the TAS lock, and its detailed proof is in Fig. 15, which will be explained later. The acquirement of L1 may or may not consume a level-1 $\blacklozenge$-token, depending on whether L2 is free or not (see p_1 in Fig. 14). If L2 is free, line 1 is a *Lock1* action, which consumes a token. Otherwise it is a *Lock0* action and the token is not consumed, allowing the thread t to roll back and do *Lock1* later. Then we stabilize the assertion. For the left branch, when an environment thread acquires L2, i.e., the interference is at level 2, the thread t could re-gain the level-1 $\blacklozenge$-token, resulting in the right branch of the assertion. Stabilizing the right branch gives us the whole p_1 too. Thus p_1 is stable.

Then thread t tests L2 at line 2 in Fig. 14. When r is not 0, thread t goes into the loop at line 3. The Q for the loop is defined at the bottom of Fig. 14, which says that thread t could terminate the loop when L1 is not acquired by the environment and L2 is free. Before line 3, we give the thread one $\lozenge$-token for the loop (applying the HIDE-$\lozenge$ rule). The token is consumed at the beginning of an iteration when the above Q holds. Thus, the loop body (from line 4 to line 6) is verified with the precondition p_2. Note the thread still has one $\lozenge$-token if L2 is not available, because the loop does not consume the token if Q does not hold. The token will be consumed at the next round when L2 is free. On the other hand, stabilizing the left branch of the above assertion p_2 just gives us the whole p_2: When an environment thread acquires L2, thread t could re-gain a $\lozenge$-token.

Besides, the definite progress $R, G\colon \mathcal{D} \xrightarrow{f} Q$ is verified as follows. When thread t is blocked (i.e., Q does not hold), there is a queue of definite actions of the environment threads. The length of the queue is at most 2, as shown by f defined below:

$$f_{\mathtt{t}}(\mathfrak{S}) \overset{\text{def}}{=} \left\{ \begin{array}{ll} 2 & \text{if } \mathfrak{S} \models \mathsf{envLocked_t}(\mathtt{L2}) * \mathsf{true} \\ 1 & \text{if } \mathfrak{S} \models \mathsf{envLocked_t}(\mathtt{L1}) * \mathsf{unlocked}(\mathtt{L2}) \\ 0 & \text{if } \mathfrak{S} \models Q \end{array} \right.$$

When the environment thread t′ holding L2 releases the lock (i.e., it does $\mathcal{D}2_{\mathtt{t}'}$), the queue becomes shorter. Thread t only needs to wait for the environment thread to release L1.

Acquirements of `L1` in the rollback example. Finally we discuss the proof of the implementation of `lock L1` in Fig. 15. Here we use the same Q in Fig. 14 to verify the loop. That is, we think the thread is "blocked" if `L2` is not available. Initially we give one $\lozenge$-token for the loop. Depending on whether Q holds or not, there are three cases (p_{01}, p_{02} and p_{03}) when we enter the loop. For the case p_{01}, the loop consumes the $\lozenge$-token because Q holds. For the other two cases (p_{02} and p_{03}), Q does not hold and the token is kept. Note that stabilizing p_{01} results in p_{03} when the environment acquires `L1`: Since `L2` is free, the environment action is a level-1 delaying action *Lock1*, allowing the thread to re-gain the $\lozenge$-token.

For line 3, if `b` is true, we know p_{01} or p_{02} holds before the line. Depending on whether `L2` is available or not, the action may or may not consume a level-1 $\blacklozenge$-token, following the same argument as in line 1 in Fig. 14. If `b` is false, then p_{03} holds before the line. Stabilizing this case results in the right branch of the postcondition of line 3, with a $\lozenge$-token for the next round of loop.

It may seem strange that for the loop we do not use a $Q' \overset{\text{def}}{=} \mathsf{locked}_{\mathsf{t}}(\texttt{L1}) \vee \mathsf{unlocked}(\texttt{L1})$, i.e., the same Q as in Fig. 12. If we use Q' instead, then the case p_{02} before line 3 cannot have the $\lozenge$-token because Q' is true in this case and the $\lozenge$-token needs to be consumed by the loop. Thus p_{02} needs to be changed to $p'_{02} \overset{\text{def}}{=} \mathsf{envLocked}_{\texttt{cid}}(\texttt{L1}) * \mathsf{notOwn}_{\texttt{cid}}(\texttt{L2})$. Stabilizing p'_{02} can no longer give us p_{03} when the environment acquires `L1`, because the acquirement action is *Lock0* instead of *Lock1* (since `L2` is not free in this case). The thread cannot re-gain the $\lozenge$-token in p_{03}, so p_{03} cannot have the $\lozenge$-token either. As a result, we no longer have a $\lozenge$-token to pay for the next round of loop if the `cas` in line 3 fails.

5. Soundness and Abstraction Theorems

Our logic LiLi is a sound proof technique for concurrent objects based on blocking algorithms, as shown by the following theorem.

Theorem 2 (Soundness). If $\mathcal{D}, R, G \vdash \{P\}\Pi : \Gamma$, then

(1) both $\Pi \preceq^{\mathsf{lin}}_{P} \Gamma$ and $\mathsf{deadlock\text{-}free}_{P}(\Pi)$ hold; and
(2) if $R \Rightarrow \lfloor R \rfloor_0$ and $G \Rightarrow \lfloor G \rfloor_0$, then $\mathsf{starvation\text{-}free}_{P}(\Pi)$ holds.

Here $\Pi \preceq^{\mathsf{lin}}_{P} \Gamma$ describes linearizability of the object Π. Informally it says that Π has the same effects as the *atomic* operations of Γ (assuming the initial object states satisfy P). The formal definition is standard [15] and omitted here. $\mathsf{deadlock\text{-}free}_{P}(\Pi)$ and $\mathsf{starvation\text{-}free}_{P}(\Pi)$ are the two progress properties defined following their informal meanings [14] (or see Sec. 1).

Theorem 2 shows that LiLi ensures linearizability and deadlock-freedom *together*, and it also ensures starvation-freedom when the rely/guarantee specification of the object satisfies certain constraints. The constraints $R \Rightarrow \lfloor R \rfloor_0$ and $G \Rightarrow \lfloor G \rfloor_0$ require R and G to specify actions of level 0 only. 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 starvation-freedom.

Abstraction. The soundness theorem shows that our logic ensures linearizability with respect to atomic operations. However, from the client code's point of view, the methods of deadlock-free objects do *not* refine atomic operations when termination is concerned. Consider the example below.

```
dfInc(); s:=1; || while (s=0) dfInc();
    INC; s:=1; || while (s=0) INC;
```

The first line shows the client code using a lock-free counter, while the second line uses an atomic counter (see Fig. 1 for the implementation of counters). Assuming `s=0` initially, it is easy to see the first program may or may not terminate, but the second one must terminate under *fair scheduling*. Therefore the first program is *not* a termination-preserving refinement of the second one. Note that if we replace `dfInc` with the starvation-free counter `sfInc`, the first program must terminate too under fair scheduling.

We propose a novel "progress-aware" object specification $\mathsf{wr}_{\texttt{l}}(\Gamma)$ for deadlock-free objects that are linearizable with respect to Γ. As defined below, $\mathsf{wr}_{\texttt{l}}(\Gamma)$ wraps the atomic operations in Γ with some auxiliary code for synchronization.

$$\mathsf{wr}_{\texttt{l}}(\Gamma)(f) \overset{\text{def}}{=} (x,\, \mathsf{wr}_{\texttt{l}}(\langle C\rangle); \textbf{return } E) \quad \text{if } \Gamma(f) = (x,\, \langle C\rangle; \textbf{return } E)$$

$\mathsf{wr}_{\texttt{l}}(\langle C\rangle) \overset{\text{def}}{=}$
```
local u1 := nondet(), u2 := nondet();
while(u1 >= 0) { lock l; unlock l; u1--; }
⟨C⟩;
while(u2 >= 0) { lock l; unlock l; u2--; }
```

Here we assume `l` is a fresh variable, i.e., $\texttt{l} \notin \mathit{fv}(\Pi, \Gamma, P)$. The wrapper function $\mathsf{wr}_{\texttt{l}}$ inserts a finite (but arbitrary) number of lock-acquire (`lock l`) and lock-release (`unlock l`) actions before and after the atomic abstract code $\langle C\rangle$. The command `u := nondet()` assigns to `u` a nondeterministic number. The lock `l` is a TAS lock (implementation shown in Fig. 1(b)). Then the progress of a thread executing $\mathsf{wr}_{\texttt{l}}(\Gamma)$ could be delayed by other threads acquiring the lock `l`. By introducing the explicit delay mechanism, the abstract specification can model the deadlock-freedom property. In our previous example, if we replace `INC` with `wr`$_{\texttt{l}}$`(INC)`, the second program may fail to terminate too, even under fair scheduling.

Before showing our abstraction theorem, we first define contextual refinement below.

Definition 3 (Contextual Refinement under Fair Scheduling).

$$\begin{aligned}\Pi \sqsubseteq_P \Pi' \ \text{ iff } \ & \forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, \Sigma_o.\ ((\sigma_o, \Sigma_o) \models P) \\ & \Longrightarrow\ \mathcal{O}_{f\omega}[\![(W, (\sigma_c, \sigma_o))]\!] \subseteq \mathcal{O}_{f\omega}[\![(W', (\sigma_c, \Sigma_o))]\!]\,;\end{aligned}$$
where $W = \textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n$ and $W' = \textbf{let } \Pi' \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n$.

Here $\mathcal{O}_{f\omega}[\![W, \mathcal{S}]\!]$ generates the full traces of externally observable events in *fair* executions starting from $(W, \mathcal{S})$. In our language in Sec. 3, only outputs (produced by **print** commands) and fault events are externally observable. Note that $\mathcal{O}_{f\omega}[\![W, \mathcal{S}]\!]$ contains only *full* execution traces (which could be infinite for non-terminating executions), therefore $\sqsubseteq$ is a termination-preserving refinement.

As an important and novel result, we show the following abstraction theorem, saying that our logic LiLi ensures the contextual refinements $\sqsubseteq$.

Theorem 4 (Progress-Aware Abstraction).
Suppose $\texttt{l} \notin \mathit{fv}(\Pi, \Gamma, P, \mathcal{D}, R, G)$. If $\mathcal{D}, R, G \vdash \{P\}\Pi : \Gamma$, then

(1) $\Pi \sqsubseteq_{\mathsf{wr}_{\texttt{l}}(P)} \mathsf{wr}_{\texttt{l}}(\Gamma)$ holds; and
(2) if $R \Rightarrow \lfloor R \rfloor_0$ and $G \Rightarrow \lfloor G \rfloor_0$, then $\Pi \sqsubseteq_P \Gamma$ holds.

where $\mathsf{wr}_{\texttt{l}}(P)$ extends the precondition P with the lock variable `l`, i.e., $\mathsf{wr}_{\texttt{l}}(P) \overset{\text{def}}{=} (P * (\texttt{l} \mapsto 0))$.

The contextual refinements $\sqsubseteq$ provide abstractions for concurrent objects under fair scheduling, which can be applied for modular verification of client code. When proving liveness properties of a client of an object under fair scheduling, we can soundly replace the concrete object implementation Π by some more abstract code. For starvation-free objects (case (2) in the theorem), the substitute is Γ, the atomic abstract operations. For deadlock-free objects (case (1) in the theorem), the substitute is $\mathsf{wr}_{\texttt{l}}(\Gamma)$, where the atomic abstract operations Γ are wrapped with auxiliary code for synchronization. In this paper we do not discuss the verification of clients.

More details about proofs. Due to splace limit, we omit definitions of some key concepts, e.g., linearizability, deadlock-freedom and starvation-freedom. They are mostly standard and are presented in the TR [22]. The proofs of Theorems 2 and 4 are shown in detail in the TR too. Here we only give a brief overview about the structure of the proofs.

	non-delay		*delay*
non-blocking	wait-freedom	$\Rightarrow$	lock-freedom
	$\Downarrow$		$\Downarrow$
blocking	starvation-freedom	$\Rightarrow$	deadlock-freedom

Figure 16. Progress properties of concurrent objects.

To prove Theorem 4, we introduce a termination-preserving simulation, which extends previous work [24] to reason about blocking and delay. LiLi ensures that the concrete implementation Π is simulated by the abstract specification (Γ for starvation-free objects and $\mathsf{wr_1}(\Gamma)$ for deadlock-free ones). Then we prove the simulation ensures the contextual refinement $\sqsubseteq$.

We also establish the equivalence between the contextual refinements and the combination of linearizability and deadlock-freedom/starvation-freedom, as in Liang et al.'s previous work [23]. Then Theorem 2 follows from these equivalence results and Theorem 4.

6. On Lock-Freedom and Wait-Freedom

As a program logic for concurrent objects under fair scheduling, LiLi unifies the verification of linearizability, starvation-freedom and deadlock-freedom. It has been applied to verify objects with blocking synchronization (i.e., mutual exclusion locks).

LiLi can also be applied to verify non-blocking objects. For non-blocking objects, wait-freedom and lock-freedom are two commonly accepted progress criteria, which require method-wise progress and whole-system progress respectively. Then, under fair scheduling, wait-freedom and lock-freedom are degraded to starvation-freedom and deadlock-freedom, respectively.

Fig. 16 shows the relationships among all the four progress properties (where "$\Rightarrow$" represents implications). We sort them in two dimensions: *blocking* and *delay* (their difference has been explained in Sec. 2.2.1). Starvation-free or deadlock-free objects allow a thread to be blocked, and lock-free and deadlock-free objects permit delay.

Our logic LiLi handles blocking by definite actions, and supports delay by $\blacklozenge$-tokens and multi-level actions. By ignoring either or both features, it can be instantiated to verify objects with any of the four progress properties in Fig. 16.

To verify lock-free objects, we instantiate the definite actions $\mathcal{D}$ to be $\mathsf{false} \leadsto \mathsf{true}$, and use only the supports for delay. Then a thread cannot rely on the environment threads' $\mathcal{D}$, meaning that it is never blocked. The WHL rule in Fig. 7 is reduced to requiring that the loop terminates (the $\lozenge$-tokens decrease at each iteration) unless being delayed by the environment. The definite progress condition $J \Rightarrow (R, G\colon \mathcal{D} \xrightarrow{f} Q)$ could trivially hold by setting both Q and J to be true and f to be a constant function.

To verify wait-free objects, besides instantiating $\mathcal{D}$ as $\mathsf{false} \leadsto \mathsf{true}$, we also require R and G to specify actions of level 0 only, as in Theorem 2(2). The instantiation results in the logic rules disallowing both blocking and delay, so we know every method would terminate regardless of the environment interference.

In fact, Liang et al.'s program logic rules [24] for lock-free algorithms can be viewed as a specialization of LiLi. Thus all the examples verified in their work can also be verified in LiLi.

7. More Examples

We have seen a few small examples showing the use of LiLi. Below we give an overview of other blocking algorithms we have verified. Their proofs are in TR [22].

- *Coarse-grained synchronization.* The easiest way to implement a concurrent object is using a single lock to protect all the object data. Our logic can be applied to such an object. As an example, we verified the counter with various lock implementations [13, 25], including ticket locks, Anderson array-based queue locks, CLH list-based queue locks, MCS list-based queue locks, and TAS locks. We show that the coarse-grained object with ticket locks or queue locks is starvation-free, and it is deadlock-free with TAS locks.
- *Fine-grained and optimistic synchronization.* As examples with more permissive locking scheme, we verified Michael-Scott two-lock queues [26], lock-coupling lists [13], optimistic lists [13], and lazy lists [11]. We show that the two-lock queues and the lock-coupling lists are starvation-free if all their locks are implemented using ticket locks, and they are deadlock-free if their locks are TAS locks. The optimistic lists and the lazy lists have rollback mechanisms, and we prove they are deadlock-free.

To the best of our knowledge, we are the first to formally verify the starvation-freedom of lock-coupling lists and the deadlock-freedom of optimistic lists and lazy lists.

Optimistic lists. Below we verify the optimistic list-based implementation in Fig. 2(a) of a mutable set data structure. The algorithm has operations `add`, which adds an element to the set, and `rmv`, which removes an element from the set. Fig. 17 shows the code and the proof outline for `rmv`.

We have informally explained the idea of the algorithm in Sec. 2.3.4. To verify its progress in LiLi, we need to recognize the delaying actions, specify them in rely and guarantee conditions with appropriate levels, define the definite actions, and finally prove the termination of loops following the WHL rule.

Following the earlier linearizability proofs in RGSep [31], the basic actions of a thread include the lock acquire (line 4) and release actions (lines 6 and 12), and the *Add* and *Rmv* actions (lines 8-11) that insert and delete nodes from the list respectively. Since we use TAS locks here, acquirements of a lock will delay other threads competing for the same lock. Thus lock acquirements are delaying actions, as illustrated in Sec. 4.3.2. Also, the *Add* and *Rmv* actions may cause the failure of the validation (at line 5) in other threads. The failed validation will further cause the threads to roll back and to acquire the locks again. Therefore the *Add* and *Rmv* actions are also delaying actions that may lead to more lock acquirements. In our rely and guarantee specifications, the *Add* and *Rmv* actions are level-2 delaying actions, while lock acquirements are at level 1.

Next we define the definite actions $\mathcal{D}$ that a blocked thread may wait for. Since a thread is blocked only if the lock it tries to acquire is unavailable, we only need to specify in $\mathcal{D}$ the various scenarios under which the lock release would definitely happen. The definitions are omitted here.

We also need to find a metric f to prove the definite progress condition in the WHL rule. We define f as the number of all the locked nodes, including those on the list and those that have been removed from the list but have not been unlocked yet. It is a conservative upper bound of the length of the queue of definite actions that a blocked thread is waiting for. It is easy to check that every definite action $\mathcal{D}$ makes the metric to decrease, and that a thread is unblocked to acquire the lock when the metric becomes 0.

In Fig. 17, the precondition is given $\blacklozenge(1, 2)$, two level-1 $\blacklozenge$-tokens for locking two adjacent nodes, and one level-2 $\blacklozenge$-token for doing *Rmv*. We apply the (HIDE-$\lozenge$) rule and assign one $\lozenge$-token to the loop at lines 2-7, so the loop should terminate in one round if it is not delayed by the environment.

A round of loop is started at the cost of the $\lozenge$-token. The code `find` at line 3 traverses the list. After line 3, `p` and `c` may be valid: both of them are on the list and `p.next` is `c`. However, if the environment updates the list by the level-2 delaying actions *Add* or *Rmv*, the two nodes `p` and `c` may no longer satisfy `valid`. In this case, `invalid(p, c)` holds, and the current thread could gain two more

```
rmv(int e) {
      {◆(1,2) ∧ arem(RMV(e)) ∧ ...}
  1   local b := false, p, c, n;
      {¬b ∧ ◆(1,2) ∧ ◇ ∧ ... ∨ b ∧ ...}
  2   while (!b) {
        {◆(1,2) ∧ ...}
  3     (p, c) := find(e); // a loop of list traversal
        {valid(p,c) ∧ ◆(1,2) ∧ ... ∨ invalid(p,c) ∧ ◆(1,4) ∧ ◇ ∧ ...}
  4     lock p; lock c;
        {valid(p,c) ∧ ◆(1,0) ∧ ... ∨ invalid(p,c) ∧ ◆(1,2) ∧ ◇ ∧ ...}
  5     b := validate(p, c); // a loop of list traversal
  6     if (!b) { unlock c; unlock p; }
        {b ∧ valid(p,c) ∧ ◆(1,0) ∧ ... ∨ ¬b ∧ ◆(1,2) ∧ ◇ ∧ ...}
  7   }
      {valid(p,c) ∧ ◆(1,0) ∧ arem(RMV(e)) ∧ ...}
  8   if (c.data = e) {
  9     n := c.next;
        {valid(p,c,e,n) ∧ ◆(1,0) ∧ arem(RMV(e)) ∧ ...}
 10     < p.next := n;  gn := gn ∪ {c} >; // LP
        {valid(p,n) ∧ arem(skip) ∧ ...}
 11   }
 12   unlock c; unlock p;
      {arem(skip) ∧ ...}
}
```

Figure 17. Proofs for optimistic lists (with auxiliary code in gray).

level-1 ◆-tokens and one more ◇-token, allowing it to roll back and re-lock the nodes in a new round.

At line 4, `lock p` and `lock c` consume two level-1 ◆-tokens respectively. The validation at line 5 succeeds in a valid state, and fails in an invalid state. Thus we can re-establish the loop invariant after line 6.

Lines 8-11 perform the node removal. Line 10 is the linearization point (LP in the figure), at which we fulfill the abstract atomic operation `RMV(e)`. Afterwards, the remaining abstract code becomes **skip**. To help specify the shared state, in line 10 we introduce an auxiliary variable gn to collect the locations of removed nodes.

Due to space limit, here we only give a brief overview of the proofs and omit many details, including the specifications of rely and guarantee conditions, definite actions, and the proofs of the implementation of `find` (line 3), `validate` (line 5) and `lock` (line 4). The full specifications and proofs are given in our TR [22].

8. Related Work and Conclusion

Using rely-guarantee style logics to verify liveness properties can date back to work by Stark [28], Stølen [29], Abadi and Lamport [1] and Xu et al. [33]. Among them the most closely related work is the fair termination rule for while loops proposed by Stølen [29], based on an idea of wait conditions. His rule requires each iteration to descend if the wait condition P_w holds once in the round. P_w is comparable to $\neg Q$ in our WHL rule in Fig. 7. But it is difficult to specify P_w which is part of the global interface of a thread, while our Q can be constructed on-the-fly for each loop. Also it is difficult to construct the well-founded order when $\neg P_w$ is not stable (e.g., as in the TAS lock). We address the problem with the token transfer idea. Besides, his rule does not support starvation-freedom verification.

Gotsman et al. [10] propose a rely-guarantee-style logic to verify non-blocking algorithms. They allow R and G to specify certain types of liveness properties in temporal logic assertions, and do layered proofs iteratively in multiple rounds to break circular reasoning. Afterwards Hoffmann et al. [16] propose the token-transfer idea to handle delays in lock-free algorithms. Their approach can be viewed as giving relatively lightweight guidelines (without the need of multi-round reasoning) to discharge the temporal obligations for lock-freedom verification. Liang et al. [24] then apply similar ideas in refinement verification. Their logic can verify linearizability and lock-freedom together. In LiLi, the use of stratified ◆-tokens generalizes their token-transfer approaches to support delays and rollbacks for deadlock-free objects. Also we propose the new idea of definite actions as a specific guideline to support blocking for progress verification under fair scheduling.

Recently, da Rocha Pinto et al. [5] take a different approach to handle delay. They verify total correctness of non-blocking programs by explicitly specifying the number of delaying actions that the environment can do. As we explained, blocking and delay are two different kinds of interference causing non-termination, both of which are now handled in LiLi.

Jacobs et al. [17] also design logic rules for total correctness. They prevent deadlock by global wait orders (proposed by Leino et al. [20] to prove safety properties), where they need a global function mapping locks to levels. It is unclear if their rules can be applied to algorithms with dynamic locking and rollbacks, such as the list algorithms verified with LiLi. Besides, the idea of wait orders relies on built-in locks, which is ill-suited for object verification since it is often difficult to identify a particular field in the object as a lock.

Boström and Müller [3] extend the approach of global wait orders to verify finite blocking in non-terminating programs. They propose a notion of obligations which are like our definite actions $\mathcal{D}$. But they still do not support starvation-freedom verification. Here we propose the definite progress condition to also ensure the termination of a thread if it is unblocked infinitely often.

Filipović et al. [8] first show the equivalence between linearizability and a contextual refinement. Gotsman and Yang [9] suggest a connection between lock-freedom and a termination-sensitive contextual refinement. Afterwards Liang et al. [23] formulate several contextual refinements, each of which can characterize a liveness property of linearizable objects. However, their contextual refinements for blocking properties assume fair scheduling at the concrete level only, which lack transitivity. In this paper, we unify deadlock-freedom and starvation-freedom with the contextual refinement $\sqsubseteq$ (see Def. 3) which gives us the novel Abstraction Theorem (Thm. 4) to support modular reasoning about client code.

Back and Xu [2] and Henzinger et al. [12] propose simulations to verify refinement under fair scheduling. Their simulations are not thread-local, and there is no program logic given.

There is also plenty of work for liveness verification based on temporal logics and model checking. Temporal reasoning allows one to verify progress properties in a unified and general way, but it provides less guidance on how to discharge the proof obligations. Our logic rules are based on program structures and enforce specific patterns (e.g., definite actions and tokens) to guide liveness proofs.

Conclusion and future work. We propose LiLi to verify linearizability and starvation-freedom/deadlock-freedom of concurrent objects. It is the first program logic that supports progress verification of blocking algorithms. We have applied it to verify several non-trivial algorithms, including lock-coupling lists, optimistic lists and lazy lists. In the future, we would like to further test its applicability with more examples, such as tree algorithms which perform rotation by fine-grained locking. We also hope to mechanize LiLi and develop tools to automate the verification process.

Acknowledgments

We would like to thank Zhong Shao and 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. 61229201, 61379039, 91318301 and 61502442.

References

[1] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–535, 1995.

[2] R. Back and Q. Xu. Refinement of fair action systems. *Acta Inf.*, 35 (2):131–165, 1998.

[3] P. Boström and P. Müller. Modular verification of finite blocking in non-terminating programs. In *ECOOP*, pages 639–663, 2015.

[4] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.

[5] P. da Rocha Pinto, T. Dinsdale-Young, P. Gardner, and J. Sutherland. Modular termination verification for non-blocking concurrency, 2015. Manuscript.

[6] J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *ACM Trans. Program. Lang. Syst.*, 33(1):4:1–4:43, 2011.

[7] X. Feng. Local rely-guarantee reasoning. In *POPL*, pages 315–327, 2009.

[8] I. Filipović, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010.

[9] A. Gotsman and H. Yang. Liveness-preserving atomicity abstraction. In *ICALP*, pages 453–465, 2011.

[10] A. Gotsman, B. Cook, M. J. Parkinson, and V. Vafeiadis. Proving that non-blocking algorithms don't block. In *POPL*, pages 16–28, 2009.

[11] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer III, and N. Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, pages 3–16, 2005.

[12] T. A. Henzinger, O. Kupferman, and S. K. Rajamani. Fair simulation. *Inf. Comput.*, 173(1):64–81, 2002.

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

[14] M. Herlihy and N. Shavit. On the nature of progress. In *OPODIS*, pages 313–328, 2011.

[15] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[16] J. Hoffmann, M. Marmar, and Z. Shao. Quantitative reasoning for proving lock-freedom. In *LICS*, pages 124–133, 2013.

[17] B. Jacobs, D. Bosnacki, and R. Kuiper. Modular termination verification. In *ECOOP*, pages 664–688, 2015.

[18] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[19] K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In *ESOP*, pages 378–393, 2009.

[20] K. R. M. Leino, P. Müller, and J. Smans. Deadlock-free channels and locks. In *ESOP*, pages 407–426, 2010.

[21] H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, pages 459–470, 2013.

[22] H. Liang and X. Feng. A program logic for concurrent objects under fair scheduling (technical report), 2015. `http://kyhcs.ustcsz.edu.cn/relconcur/lili`.

[23] H. Liang, J. Hoffmann, X. Feng, and Z. Shao. Characterizing progress properties of concurrent objects via contextual refinements. In *CONCUR*, pages 227–241, 2013.

[24] H. Liang, X. Feng, and Z. Shao. Compositional verification of termination-preserving refinement of concurrent programs. In *CSL-LICS*, pages 65:1–65:10, 2014.

[25] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[26] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.

[27] M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logics. In *LICS*, pages 137–146, 2006.

[28] E. W. Stark. A proof technique for rely/guarantee properties. In *FSTTCS*, pages 369–391, 1985.

[29] K. Stølen. Shared-state design modulo weak and strong process fairness. In *FORTE*, pages 479–498, 1992.

[30] A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, pages 377–390, 2013.

[31] V. Vafeiadis. Modular fine-grained concurrency verification, 2008. PhD Thesis.

[32] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for java libraries. In *ECOOP*, pages 602–629, 2005.

[33] Q. Xu, W. P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Asp. Comput.*, 9(2):149–174, 1997.