A Program Logic for Concurrent Objects under Fair Scheduling
(Extended Version)

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 xgfx@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.

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.

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] [23]), 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] [29] [30]), or aim for non-blocking progress properties (e.g., [5] [10] [16] [23]), 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] [31]), 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. We discuss deadlock-freedom 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 Linearity and Liveliness). 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 [23], 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][23]. 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 [24] 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.

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 lock (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][23][30], one can extend a rely-guarantee logic to verify linearizability. We use an assertion $arem(C)$ 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 $R, G \vdash \{P \land arem(INC)\}dfInc\{Q \land arem(skip)\}$. 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 $arem(skip)$ 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 deadlock-breaking example.

```
DL-12:    DL-21:
lock L1;  lock L1;  lock L2;  lock L2;
lock L2;  unlock L2; unlock L1;
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. 3(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 \texttt{cas} command in the code of \\texttt{dfInc} and have to loop at least one more round before termination. This may happen infinitely many times, causing non-termination of the \\texttt{dfInc} method on the left. In this case we say the progress of the left method is delayed by its environment’s successful acquisition 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 \\texttt{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. 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 relying-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 $L_1$ (line 1) and then tests whether $L_2$ is available (line 2). If the test fails, the thread rolls back. It releases $L_1$ (line 4), and then repeats the process of acquiring $L_1$ (line 5) and testing $L_2$ (line 6). Suppose another thread $t’$ does the opposite: repeatedly acquiring $L_1$ and testing $L_1$. In this example the acquirement of $L_2$ by $t’$ may cause $t$ to fail its test of the availability of $L_2$. The test could have succeeded if $t’$ did not interfere, so $t’$ delays $t$. Conversely, the acquirement of $L_1$ 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 \\texttt{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.

\begin{verbatim}
x := 1; while (y = 1) skip; y := 0;
x := 0; while (x = 1) skip; y := 1;
\end{verbatim}

Moreover, sometimes we need to look into the lock implementation to prove starvation-freedom. For instance, the \\texttt{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 \\texttt{dfInc} 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 dependency 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 synchronization 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 \[16,23\] requires each round of the loop to consume resources called tokens. The rule for loops is in the following form:

\[
P \land B \Rightarrow P' \land \circledast \quad R, G \vdash \{P'\}C(P)\]

Here \(\circledast\) 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 \(\text{sfInc}\) in Fig. 1(c). It uses the shared variables \(\text{owner}\) and \(\text{next}\) to guarantee the first-come-first-served property of the lock. Initially \(\text{owner}\) equals \(\text{next}\). To acquire the lock, a thread atomically increments \(\text{next}\) and reads its old value to a variable \(i\) (line 2). The value of \(i\) becomes the thread’s ticket. The lock waits until \(\text{owner}\) equals its ticket value \(i\) (line 3). Finally the lock is released by incrementing \(\text{owner}\) (line 5) such that the next waiting thread (the thread with ticket \(i + 1\), if there is one) can now enter the critical section.

We see \(\text{sfInc}\) is not concerned with the circular dependency problem. Intuitively the ticket lock algorithm ensures that the threads requesting the lock always constitute a queue \(t_1, t_2, \ldots, t_n\). The head thread, \(t_1\), gets the ticket number which equals \(\text{owner}\) and can immediately acquire the lock. Once it releases the lock (by increasing \(\text{owner}\)), \(t_1\) 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 \(P_d \sim Q_d\). 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 \(\text{sfInc}\), the definite action \(P_d \sim Q_d\) of a thread can be defined as follows. \(P_d\) says that \(\text{owner}\) equals the thread’s ticket number \(i\), and \(Q_d\) says that \(\text{owner}\) has been increased to \(i + 1\). 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 \(\text{TERM}\) 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 \(t'\)) 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 \(\text{TERM}\) 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 \(t'\) 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 \(\text{sfInc}\), \(Q\) is defined as \((i = \text{owner})\) and the metric \(E\) is \((i - \text{owner})\). Whenever an environment thread \(t'\) finishes a definite action by releasing the lock, it increases \(\text{owner}\), 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 \(\text{dfInc}\) in Fig. 1(b) implements a deadlock-free counter using the \(\text{TAS}\) 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 \(t'\) may preempt \(t\) and do a successful case. Then thread \(t\) is blocked and waits for a queue of definite actions again. This delay caused by thread \(t'\) can be viewed as a queue jump in our definite-progress-based reasoning. Actually \(\text{dfInc}\) 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 \(t'\) delays \(t\) only if \(t'\) successfully acquires the lock, which allows it to eventually finish the \(\text{dfInc}\) 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 \[16,23\] 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 \(\text{token}\) 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 \(t'\) delays the progress of thread \(t\), we require \(t'\) to consume one \(\text{token}\). At the same time, thread \(t\) could increase \(\circledast\)-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 \(\text{token}\).
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 next 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 ♦-token, it seems that the thread would need an infinite number of ♦-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 ♦-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 ♦-token. Note that the ♦-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 ♦-tokens at all levels constitute a tuple (m1, ..., mk, mk+1). It is strictly descending along the dictionary order.

The stratified ♦-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.

### Syntax of the programming language.

| (MName) f ∈ String | (Expr) E, E ::= x | n | E + E | ...
|---------------------|------------------------|-----------------------------|
| (BExp) B, B ::= true | false | E = E | !B | ...
| (Instr) c, c ::= x := E | x := [E] | E := E | print(E) |
|                     | x := cons(E,...,E) | dispose(E) | ...
| (Stmt) C, C ::= skip | c | x := f(E) | return E |
|                     | end | C;C | if (B) C else C |
| (Prog) W, W ::= | skip | let Π in C ||...|C |
| (ODecl) Π, Γ ::= | f1 ∼ (x1, C1), ..., fn ∼ (xn, Cn) |

| 3. Programming Language |

Fig.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 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 commands x := [E] and E := E' reads and writes the memory cell at the location E, respectively. x := cons(E,...,E) and dispose(E) allocates and frees memory blocks respectively. The command print(E) generates externally observable events, which allows us to observe behaviors of programs. We use (C) to represent an atomic block in which C is executed atomically. We introduce an auxiliary command end to help define liveness properties (see Sec. 3). It is appended at the end of each thread when the program W starts to run, and is not supposed to be used directly by programmers. Executing it generates a special event, which marks the termination of the corresponding thread.

Note that we use one language for both the concrete implementation at the low level and the abstract specification at the high level. We introduce two sets of symbols for each syntactic category (e.g., Π and Γ for objects).

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 command. Also we assume there is no nested method invocation. For the abstract object specification Γ, each method body is an atomic operation (C) (ahead of the return command), and executing the code is always safe and never blocks.

The model of program states S is defined in Fig.4. To ensure that the client code does not touch the object state, in S we separate the states accessed by clients (σc) and by the object (σo). Execution of programs is modelled as a labelled transition system (W,S) → (W',S') in Fig.5. The event e is defined in Fig.4 (t.obj) and (t.clt) represent an execution step by the thread t inside and outside the object method, respectively. (t.f,n) and (t.ret,n) record the invocation and the return of an object respectively. The event (t.obj.abort) or (t.clt.abort) is generated when the thread t aborts inside (or outside) the object method body. The output event (t.out,n) is generated by the print(E) command. The event (t.edge) marks the termination of the thread t, generated by the end command. The event (spawn,n) is used to record the number n of threads in the program, which is helpful to define fair executions (see Sec. 3). We use T to represent an event trace, a (possibly infinite) sequence of events.

**Externally observable events.** We treat (t.out,n), (t.obj.abort) and (t.clt.abort) as externally observable events, and we use get_obsv(T) to represent the subsequence of T consisting of externally observable events only.

### 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 D,R,G ⊢ {P}Π:Γ. (The OBJ rule for this judgment is given in Fig.9 and will be explained later.) To verify an object Π, we give a corresponding object specification Γ (see
\[
\begin{align*}
\text{(let } \Pi \text{ in skip || skip, } S) & \rightarrow (\text{skip, } S) \\
\text{(let } \Pi \text{ in } C_1 \ldots C_n || C_n, (\sigma_c, \sigma_o, K(i))) & \rightarrow \text{ abort} \\
\end{align*}
\]

(a) program transitions

\[
\begin{align*}
\Pi(f) = (y, C) \\
\text{and } (s, \ell, C, \sigma_o, o) \quad \text{or } (t, n, C, \sigma_o, o) \\
\end{align*}
\]

\[
\begin{align*}
\kappa = (s_1, x, C) \\
\text{and } (s, \ell, C, \sigma_o, o) \\
\end{align*}
\]

\[
\begin{align*}
E[x := f(E)], ((s_c, h_c), (s_o, o)) \rightarrow \text{ undefined} \\
\text{and } (s_c, h_c, (s_o, o)) \\
\end{align*}
\]

\[
\begin{align*}
E[\text{return } E], ((s_c, h_c), (s_o, o)) \rightarrow \text{ undefined} \\
\text{and } (s_c, h_c, (s_o, o)) \\
\end{align*}
\]

(b) thread transitions

\[
\begin{align*}
(C, \sigma) \rightarrow^* \text{(skip, } \sigma') \\
(C, \sigma) \rightarrow^* \text{(E[skip], } \sigma') \\
(C, \sigma) \rightarrow^* \text{ abort} \\
\end{align*}
\]

(c) local thread transitions

\[
\begin{align*}
(\text{ThrdID}) \\
(\text{ExecCtx}) \\
(\text{Store}) \\
(\text{Mem}) \\
(\text{ThrdPool}) \\
(\text{PState}) \\
(\text{Ev}t) \\
(\text{ETrace})
\end{align*}
\]

Figure 4. States and event traces.

Fig 5, 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 setting. Here P is a relational assertion that specifies the consistency relation between the concrete data elements 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 actions D is a special form of state transitions used for progress reasoning. The definitions of P, R, G and D are shown in Sec. 3.7

Note LiLi is a logic for concurrent objects I 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” [26], we assume program variables are either thread-local or read-only. The full version of LiLi (see Appendix A) extends the more advanced Rely-Guarantee-based logic LRG [17] to support dynamic allocation and ownership transfer. It also drops the assumption about program variables.

\[
\begin{align*}
\text{(RelAssn)} \\
\text{(RelAct)} \\
\text{(DAct)} \\
\text{(FullAssn)}
\end{align*}
\]

Figure 6. Syntax of the assertion language.

4.1 Assertions

We define assertions in Fig 6. The relational state assertions P and Q specify the relationship between the concrete state σ and the abstract state Σ. Here we use s and 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 E to represent the pair of states (σ, Σ), as defined in Fig. 7.

Figure 7(a) defines semantics of state assertions. The boolean expression B holds if it evaluates to true at the combined store of s and s. emp describes empty heaps. The assertion E1 → E2 specifies a singleton heap at the concrete level with the value of the expression E2 stored at the location E1. Its counterpart for an abstract level heap is represented as E1 ⇒ E2. Semantics of separating conjunction P \& Q is similar as in separation logic, except that it is now lifted to relational states E. The disjoint union of the two relational states is defined at the top of the figure. Semantics of the assertion \([P]_a\) will be defined latter (see Fig. 7(c)).

Rely/guarantee assertions R and G specify the transitions over the relational states E. Their semantics is defined in Fig. 7(b). The action P \& _k Q says that its initial relational states satisfy P and the
Therefore, to Sec. 4.3. Below we use the upper bound of the round of loops that the current thread can founded metric for our progress reasoning. Informally, for is used to stratify actions that may delay the progress of other threads.

We further instrument the relational state assertions with the and the assertion holds over disjoint parts of and the assertion is equivalent to the following:

Figure 8. Semantics of definite actions.

Definite actions. Fig. 6 also defines definite actions \( D \), whose semantics is given in Fig. 8(a). \( P \leadsto Q \) specifies the transitions where the final states satisfy \( Q \) if the initial states satisfy \( P \). It is different from \( P \times Q \) in that \( P \leadsto Q \) does not restrict the transitions if initially \( P \) does not hold. Consider the following example \( D_3 \).

\( D_3 \) 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 \( D_1 \land D_2 \) is useful for enumerating definite actions. For instance, when the program uses two locks L1 and L2, the definite action \( D \) of the whole program is usually in the form of \( D_1 \land D_2 \), where \( D_1 \) and \( D_2 \) specify L1 and L2 respectively.

We define some useful syntactic sugars in Fig. 8(b). The state assertion \( \text{Enabled}(D) \) takes the guard condition of \( D \). We use \( \text{Enabled}(D) \) to represent the state transitions of \( D \) when it is enabled at the initial state. Intuitively \( \text{Enabled}(D) \) gives us the corresponding “\( \forall \)” actions. For instance, \( (P \leadsto Q) \) is equivalent to \( P \times Q \). For the example \( D_3 \) above, \( \text{Enabled}(D_3) \) is equivalent to the following:

\[ \forall n \,(n \times n \land (n > 0)) \leadsto (n \times n + 1) \]

In addition, we define the syntactic sugar \( [D] \) as a definite action describing the preservation of \( \text{Enabled}(D) \). For the example \( D_3 \) above, \( [D_3] \) represents the following definite action:

\[ (\exists n \,(n \times n \land (n > 0))) \leadsto (\exists n \,(n \times n + 1)) \]

It specifies the transitions which keep \( x \) positive if it is positive initially. The notation \( D \leqslant D \) will be explained later in Sec. 4.2.2 Since \( D \) is a special rely/guarantee assertion, the semantics of \( D \Rightarrow D' \) follows the standard meaning of \( R \Rightarrow R' \) (or see the definition in Fig. 12).

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 \( L \mapsto t \) to represent that the lock \( L \) is acquired by the thread \( t \). We use \( P_t \) 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 \( \forall t \, P_t \Rightarrow Q_t \). The same convention applies to rely/guarantee conditions and definite actions.
4.2 Verifying Starvation-Freedom with Definite Actions

Figure 9 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 II refine its atomic specification in Γ. Starting from the initial concrete and abstract states related by P, and with the equivalent method arguments x and y at the concrete and abstract levels, the method body C must fulfill the abstract atomic operation C'. We can temporarily ignore the assertion (Ek, . . . , Er) for deadlock-freedom.

The last three premises of the OBJ rule checks the well-formedness of the specifications. The first one says the guarantee of one thread must implies the rely of all others, a standard requirement in rely/guarantee reasoning. In Fig. 10 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 ⋆-tokens. wffAct(R, D) says once a definite action Dt of a thread t is enabled it cannot be disabled by an environment step in Rt. Also such an environment step either fulfills a definite action Dt of some thread t' different from t, or preserves Enabled(Dt) too. Together with the previous premise Gt =⇒ R, this condition implies ∀t'. Gt' =⇒ [Dt] ∨ Dt'. Therefore, once Dt is enabled, the only way to disable it is to let the thread t finish the action. As an example, consider the following Dt:

\[
\text{D}_t \triangleq (L \rightarrow t) \rightarrow (L \rightarrow 0)
\]

It says that whenever a thread t acquires the lock L, it will finally release the lock. Then, wffAct(R, D) requires that when t acquires L, every step in the system either keeps L unchanged or releases L. In particular, Rt keeps L unchanged, that is, the environment cannot update the lock when L \rightarrow t.

The last premise (P =⇒ −Enabled(D)) says there cannot be enabled but unfinished definite actions when the method terminates and the object invariant P is true.

The judgment D, R, G \vdash (p \wedge \text{arem}(C')) C(q \wedge \text{arem}(\text{skip})) establishes a simulation relation between C and C', which ensures the preservation of termination when the environment guarantees the definite action D. It also ensures the execution of C guarantees D too. We explain the rules for the judgment below.

4.2.2 The WHL Rule for Loops

The WHL rule, shown in Fig. 12, is the most important rule of the logic. It establishes both of the following properties of the loop:

1. it cannot loop forever with D continuously enabled;
2. it cannot loop forever unless the current thread is waiting for some definite actions of its environment.

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

\textbf{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 ⋆-token if Enabled(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 ⋆-tokens, the loop must terminate if Enabled(D) is continuously true.

However, the last premise of the OBJ rule says Enabled(D) cannot be true if the method terminates. Therefore, Enabled(D) cannot be continuously true. Also recall the other two side conditions (wffAct(R, D) and Gt =⇒ R) of the OBJ rule guarantee that, once Enabled(D) holds, the only way to make it false is to let the current thread finish the action.

Putting all these together, we know 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 $\diamond$-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. Then the premise $(R, G : D' \rightarrow Q)$ requires the thread must be waiting for its environment to perform a finite number of definite actions.

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

1. either $\mathfrak{S} \models Q_t$,
2. or there exists $t'$ such that $t' \neq t$ and $\mathfrak{S} \models \text{Enabled}(D_{t'})$;
3. for any $t' \neq t$ and $\mathfrak{S}'$, if $(\mathfrak{S}, \mathfrak{S}', 0) \models R_t \land (D_{t'})$, then $f_t(\mathfrak{S}') < f_t(\mathfrak{S})$;
4. for any $\mathfrak{S}'$, if $(\mathfrak{S}, \mathfrak{S}', 0) \models R_t \land G_t$, then $f_t(\mathfrak{S}') \leq f_t(\mathfrak{S})$.

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

Ignoring the index 0 above, the definition says either $Q$ holds over $\mathfrak{S}$ or the definite action $D_t$ of some environment thread $t'$ is enabled. Also we must require the metric $f$ 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 $\text{Sta}(J, R, V, G)$ hold, that is, $J$ is an invariant that holds at every program point of the loop. If $(R, G : D \rightarrow Q)$ happens to satisfy the two premises, we can use $(R, G : D \rightarrow Q)$ directly as $J$, but in practice it could be easier to prove the stability requirement by finding a stronger $J$. The definition of stability $\text{Sta}(p, R)$ is given in Fig.10.

Notice in $(R, G : D' \rightarrow Q)$ we can use $D'$ instead of $D$ to simplify the proof, as long as $D' \subseteq D$ and $\text{wffAct}(R, D')$ are satisfied. The premise $D' \subseteq D$, defined in Fig.8 says $D'$ specifies a subset of definite actions in $D$. For instance, if $D$ consists of multiple definite actions and is in the form of $D_1 \land \ldots \land D_n$, $D'$ may contain only a subset of these $D_k$ (1 $\leq k \leq n$). The way to exclude in $D'$ irrelevant definite actions can simplify the proof of the condition (2) of definite progress (see Definition1).

Given the definite progress condition, we know $Q$ will be eventually true because each definite action is definite to 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 $\text{HIDE} - \diamond$ rule allows us to discard the tokens (by using $[\!]_\diamond$) 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 $\models [p]C[q]$ to represent the total correctness of $C$ in sequential separation logic. The corresponding rules are standard and elided here. We use $p \Rightarrow q$ 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.10. 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 $\text{wffAct}(R, D')$ iff $\forall t. R_t = \{D_0 \land (\forall t' \neq t. [D_{t'}] \lor D_{t'}) \land \text{Sta}(p, R) \land [(\mathfrak{S}, (u, w), C) = p \Rightarrow [\mathfrak{S'}, (u, w), C] = p \land p \Rightarrow q \iff \forall t. \sigma, \Sigma, u, w, C. \Sigma_p$.

\[(\lbrace \sigma, \Sigma, (u, w), C \rbrace = p \Rightarrow \lbrace \Sigma, \Sigma \Sigma_p \rbrace = \exists \mathfrak{S}', \mathfrak{S}'. (\lbrace \mathfrak{S}, \Sigma \Sigma_p \rbrace \rightarrow_t (\mathfrak{S}', \Sigma \Sigma_p) \land \lbrace (\sigma, \Sigma), (u, w), C' \rbrace = q) \]

**Figure 10.** Auxiliary defs. used in logic rules (simplified version).

1. $\text{local} i, o, r$;
2. $<i := \text{getAndInc}(\text{next})$;
3. $\text{ticket}[i] := \text{cid}>;$
4. $o := \text{[owner]}$; while $(i \neq o)$ do $\{ o := \text{[owner]} ; \}$
5. $[\text{atom} := i + 1]$;
6. $P_l \equiv \exists t_i, n_1, n_2. \text{lockIn}(t_i, n_1, n_2)$
7. $G_l \equiv \text{Lock}_l \lor \text{Unlock}_l$
8. $\text{atom} \equiv \exists t_i, n_1, n_2. \text{lockIn}(t_i, n_1, n_2) \land \text{lockIn}(t_i, n_1, n_2) \land \text{lockIn}(t_i, n_1, n_2) \land \text{lockIn}(t_i, n_1, n_2)$
9. $D_{\text{lockIrr}} \equiv \forall t_i, n_1, n_2. \text{lockIn}(t_i, n_1, n_2) \land \text{lockIn}(t_i, n_1, n_2) \land \text{lockIn}(t_i, n_1, n_2)$
10. $J_i \equiv \exists t_i, n_1, n_2, i, n_2, \text{locked}(t_i, n_1, n_2) \land (a \leq n_2)$
11. $Q_{\text{lockIrr}} \equiv t_i, n_2, l_2, \text{locked}(t_i, n_2, n_2) \land (a \leq n_2)$
12. $f(\mathfrak{S}) = k$ if $\mathfrak{S} \models \exists i. (\text{owner} \rightarrow n_1) \land \text{true} \land (1 - n_1 = k)$

**Figure 11.** Proofs for the ticket lock (with auxiliary code in gray).

$R$, 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.11 using our logic rules. We have informally discussed in Sec.2 the verification of the counter using a ticket lock (sfInc in Fig.11(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.11. With an empty critical section, the code functions just as skip, so Fig.11 proves it is linearizable with respect to skip. The proofs of sfInc (including its starvation-freedom and linearizability with respect to the atomic SYNC in Fig.11(c)) are given in AppendixC.

To help specify the queue of the threads requesting the lock, we introduce an auxiliary array ticket. As shown in Fig.11 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. locktl(n, n, 2) contains the auxiliary ticket array in addition to owner and next, where owner $\rightarrow n_1$ and next $\rightarrow n_2$, and if $t$ is the list of the threads recorded in ticket[n_1], ..., ticket[n_2 - 1]. We also use lockedtl(n, n, 2) for the case when $t$ is empty. That is, the lock is acquired by the first thread in $t$, while the other threads in $t$ are waiting for the lock in order. Besides, we use lockIn(t, n_1, n_2) short for locked(t, n_1, n_2) $\land$ (t $\not\in$ t). That is, the lock is “irrelevant” to the lock; it does not request the lock. The formal definitions are given in AppendixC.

The bottom of Fig.11 defines the precondition $P$ and the guarantee condition $G$ of the code. $G$ specifies the possible atomic actions of a thread $t$. $\text{Lock}_l$ adds the thread $t$ at the end of $t$ of the threads requesting the lock and increments $\text{next}$. It corresponds to line 2 of Fig.11 $\text{Unlock}_l$ releases the lock by incrementing owner and dequeuing the thread $t$ which currently holds the lock. It corresponds to line 4 of Fig.11 $\text{Rely}$ condition $R_t$ includes all the $G_{\text{rst}}$ made by the environment threads $t'$.

Next we define the definite action $D$. As we explained in Sec.2 $D_t$ requires that whenever the thread $t$ holds a lock with...
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 $J \Rightarrow (R, G' : D \rightsquigarrow Q)$ for a stable $J$. As shown in Fig. 11, we define $J_1$ to say that the thread $t$ is requesting the lock. Here $\text{locked}(h_1, t, (n_1, z, n_2))$ is similar to $\text{locked}(h_1 : t : h_2, n_1, n_2)$). It also says that the thread $t$ takes the ticket number $i$. $Q_t$ specifies the case when $h_1$ is empty (thus owner $\mapsto \lambda$). We also strengthen the guarantee condition $G$ of the loop to $\text{id}$, the identity transitions.

The metric function $f$ maps each state $S$ 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 $< \text{order}$ on natural numbers as the associated well-founded order. Then, we can verify $J \Rightarrow (R, G' : D \rightsquigarrow Q)$.

Finally, we prove that the loop terminates with one $\Diamond$-token when $Q$ holds or $D$ is enabled. Then we can conclude linearity and starvation-freedom of the ticket lock implementation in Fig. 11.

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 $\Diamond$-tokens (we use $\Diamond$ here to distinguish them from $\land$-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. 6, we index the rely/guarantee assertion $P \bowtie Q$ with a natural number $k$ and call it a level-$k$ action. We require $0 \leq k < \text{maxL}$, where $\text{maxL}$ is a predefined upper bound of all levels. Intuitively, $P \bowtie Q$ could make other threads do more actions at a level $k < \text{maxL}$. Thus $P \bowtie Q$ cannot make other threads do any more actions, i.e., it cannot delay other threads. $P \bowtie Q$ 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. 2.3.2.

To interpret the level numbers in the assertion semantics, we define $\mathcal{L}((S, \mathcal{S}'), (R, D))$ in Fig. 12, which assigns a level to a transition $(S, \mathcal{S}')$, given the specification $R$. That is, if $\mathcal{L}((S, \mathcal{S}'), (R, D)) = k$, we say $R$ views $(S, \mathcal{S}')$ as a level-$k$ transition. We let $k = \text{maxL}$ if the transition does not satisfy $R$. Given the level function, we can now define the semantics of $\mathcal{L}(R)_0$, which picks out the transitions that $R$ views as level-0 ones. For the following example $R$,

$$R \equiv (P \bowtie Q) \lor (P' \bowtie Q')$$

$\mathcal{L}(R)_0$ is equivalent to $P \bowtie Q$. Besides, $R \Rightarrow \mathcal{L}(R)_0$ means that $R$ views all states transitions as level-0 ones, thus any state transitions of $R$ should not delay the progress of other threads.

We use $(S, \mathcal{S}', i) \models R$ as a shorthand for $\mathcal{L}((S, \mathcal{S}'), (R, D)) = k$ ($k < \text{maxL}$). Then the implication $R \Rightarrow R'$ is redefined under this new interpretation, as shown in Fig. 12.

**$\Diamond$-tokens in assertions.** To ensure the progress of the whole system, we require the steps of delaying actions to pay $\Diamond$-tokens. Since we allow multi-levels of transitions to delay other threads, the $\Diamond$-tokens are stratified accordingly. We introduce the new assertion $\langle E_0, \ldots, E_j \rangle$ in Fig. 6 whose semantics is defined in Fig. 7. It says the number of each level-$j$ $\Diamond$-tokens is no less than $E_j$. Here $u$ is a sequence $(n_k, \ldots, n_1)$ recording the numbers of $\Diamond$-tokens at different levels, as defined in Fig. 12.

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. 12.

4.3.1 Inference Rules Revisited

To use the logic to verify deadlock-free objects, we need to first find 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 $\Diamond$-tokens in the precondition of each method.

Below we revisit the inference rules in Fig. 9 and explain their use of multi-level actions and $\Diamond$-tokens. In the OBJ rule, we specify in the precondition the number of $\Diamond$-tokens needed for each object method. The side condition wffAct$(R, D)$ is also redefined in Fig. 13 which is explained below.

**Decreasing $\Diamond$-tokens at the ATOM rule.** The thread loses $\Diamond$-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 $q' \Rightarrow q$, which is redefined in Fig. 13. Similar to $p \Rightarrow q$ in Fig. 10, it allows us to execute the abstract code. Now it also requires the number of $\Diamond$-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 $\Diamond$-tokens to do more loop iterations. It could also gain more level-$k'$ ($k' < k$) $\Diamond$-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 $\Diamond$-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 Sta(p, R) in Fig. 14 to reflect the
possible increasing of tokens for the thread t. We could reset w and
the number at level k<sub>′</sub> < k in t 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. 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 time on the
environment’s definite actions.

As shown in Definition 1, the definite progress condition (R, G:
D<sub>Lock</sub> Q) allows the thread to reset its metric f(Σ) for a queue jump
when the environment step is associated with level k ≥ 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
♦-token is paid by the environment thread for the delaying action.

Also the requirement wffAct(R, D) (used at the OBJ rule and
the WHL rule) should be revised to allow queue jumps. The new
definition is shown in Fig. 13. Here we allow a queue jump to disable
the definite action D of the thread at the head of the queue, so it
is not necessary to require Enabled(D) to be preserved when
the environment step is associated with level k ≥ 1.

**4.3.2 Example: Test-and-Set Locks**

In Fig. 14 we verify the test-and-set lock implementation explained
in Sec. 4.2. Like the ticket lock proofs in Sec. 4.2.4 we simplify the
code and prove it is linearizable with respect to skip. Here we omit
the assertion aren't(skip) at each line in the proof, and focus on
proving deadlock-freedom of the code.

As defined at the top of Fig. 14 the action Lock<sub>′</sub> (corresponding to
the successful cas at line 3) is at level 1, which may delay other
threads trying to acquire the lock. The Unlock<sub>′</sub> action is at level 0,
which cannot delay other threads. Also the precondition is given a
♦-token, which is required to pay for the Lock<sub>′</sub> action.

The definite action 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 D in the OBJ rule,
e.g., wffAct(R, D), are satisfied.

R, G : D<sub>Lock</sub> Q specifies the queue of definite actions which now
contains at most one environment thread. That is, the metric f(Σ)
is 1 if the lock is not available, and is 0 otherwise. When an
environment thread t<sup>′</sup> cuts in line by acquiring the lock when the
lock is free, the current thread t has to wait for D<sub>Lock</sub> before t itself
progresses. Thus in R, G : D<sub>Lock</sub> Q the current thread t can reset its
metric f(Σ) when its environment acquires the lock.

The detailed proof at the bottom of Fig. 14 shows the changes of
tokens. We give the current thread one ♦-token (using the HIDE-♦
rule) to do its loop at lines 2-4. It consumes this ♦-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 ♦-token.

Next we stabilize p. For the left branch, if an environment thread
T<sup>′</sup> acquires the lock, which is a delaying action Lock<sub>′</sub>, we let the
current thread possess a ♦-token. The resulting state just satisfies the
right branch of p. Thus p is already stable.

The current thread pays its ♦-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 case fails, the thread still has a ♦
to acquire the lock in the future and ♦ 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 acquirments are delaying actions. Here we have
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 ♦-tokens in two levels. Acquirments of
L2 are at level 2, defined as Lock2 in Fig. 15 which may trigger
rollbacks and more acquirments of L1. Acquirments 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, acquirments 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 acquirments
of L1 as delaying actions would require t to pay ♦-tokens infinitely
often, and consequently require an infinite number of ♦-tokens be
assigned to the method at the beginning, which is impossible. To
address the problem, we define in Fig. 15 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 acquirment of L1 as a non-delaying action Lock0 at level 0.
10 unlock L1;

Depending on whether notOwn lock L1 at line 01

release the two locks respectively. As shown in Fig. 15, the
notations in LRG say a

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

$$
\{ \text{notOwn}_{\text{cid}}(L1) \land \text{notOwn}_{\text{cid}}(L2) \land \orw (1,1) \} \\
1 \text{ lock L1;}

\text{p}_1 \seteq \begin{cases}
\text{locked}_{\text{cid}}(L1) \land (\text{locked}(L2) \land \orw (1,1)) \\
\lor \text{envLocked}_{\text{cid}}(L2) \land \orw (1,1))
\end{cases}

2 \text{ local } r \seteq L2:

\text{locked}_{\text{cid}}(L1) \land \text{notOwn}_{\text{cid}}(L2)

\land \begin{cases}
(r = 0) \lor \orw (1,0) \lor \land (r 
eq 0) \lor \orw (1,1) \land \orw (1,0)
\end{cases}

3 \text{ while } (r 
eq 0) \{ \\
\text{p}_2 \seteq \begin{cases}
\text{locked}_{\text{cid}}(L1)
\lor (\text{locked}(L2) \land \text{notOwn}_{\text{cid}}(L2) \land \orw (1,1))
\end{cases}
\}

4 \text{ unlock L1;}

5 \text{ lock L1;}

\text{locked}_{\text{cid}}(L1) \land (\text{locked}(L2) \land \orw (1,0))

\land \begin{cases}
\text{envLocked}_{\text{cid}}(L2) \land \orw (1,1) \land \orw (1,0)
\end{cases}

6 \text{ r } \seteq L2:

\text{locked}_{\text{cid}}(L1) \land \text{notOwn}_{\text{cid}}(L2)

\land \begin{cases}
(r = 0) \lor \orw (1,0) \lor \land (r = 1) \land \orw (1,1) \land \orw (1,0)
\end{cases}

7 \}

\text{locked}_{\text{cid}}(L1) \land \text{notOwn}_{\text{cid}}(L2) \land \orw (1,0)

8 \text{ lock L2;}

\text{locked}_{\text{cid}}(L1) \land \text{locked}_{\text{cid}}(L2)

\land \begin{cases}
\text{notOwn}_{\text{cid}}(L1) \land \text{notOwn}_{\text{cid}}(L2)
\end{cases}

9 \text{ unlock L2;}

10 \text{ unlock L1;}

\text{Q}_p \seteq \begin{cases}
(\text{locked}(L1) \land \text{unlocked}(L1)) \land \text{unlocked}(L2)
\end{cases}

\}

\begin{align*}
&\text{Fig. 15. Multi-level actions for the example in Fig. 2(b).} \\
&\text{Fig. 16. Proof outline for the rollback example in Fig. 2(b).}
\end{align*}

To simplify the presentation, the definitions in Fig. 15 follow the notations in LRG, using "\$P\$" to mean that the actions on the irrelevant part P of the states are identity transitions.

**Definite actions.** There are two kinds of definite actions, which release the two locks respectively. As shown in Fig. 13, D2 says a thread holding L2 eventually releases it, regardless of the status of the lock L1. D1 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. 16, when thread t starts the method, it is given \(\orw (1,1)\), where the level-2 \(\orw\) token is for doing Lock2 and the level-1 \(\orw\) token is for Lock1. The assertions notOwn is defined similarly as in Fig. 14.

Lock1 at line 1 is implemented using the TAS lock, and its detailed proof is in Fig. 17, which will be explained later. The acquirement of L1 may or may not consume a level-1 \(\orw\) token, depending on whether L2 is free or not (see p1 in Fig. 19). If L2 is free, line 1 is a Lock1 action, which consumes a token. Otherwise it is a Lock2 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 \(\orw\) 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. 16. 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. 16 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 \(\orw\)-token for the loop (applying the HIDE-\(\orw\) 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 p2. Note the thread still has one \(\orw\)-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 p3 just gives us the whole p2: When an environment thread acquires L2, thread t could re-gain a \(\orw\)-token.

Besides, the definite progress R, G: \(\mathcal{D} \cup Q\) 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:

\[
f_t(\mathcal{S}) \seteq \begin{cases}
2 & \text{if } \mathcal{S} = \text{envLocked}_{\text{cid}}(L2) \land \text{true} \\
1 & \text{if } \mathcal{S} = \text{envLocked}_{\text{cid}}(L1) \land \text{unlocked}(L2) \\
0 & \text{if } \mathcal{S} = Q
\end{cases}
\]

When the environment thread t holding L2 releases the lock (i.e., it does \(\mathcal{D}_2\)), 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. 17. Here we use the same Q in Fig. 16 to verify the loop. That is, we think the thread is "blocked" if L2 is not available. Initially we give one \(\orw\)-token for the loop. Depending on whether Q holds or not, there are three cases (p01, p02 and p03) when we enter the loop. For the case p01, the loop consumes the \(\orw\)-token because Q holds. For the other two cases (p02 and p03), Q does not hold and the token is kept. Note that stabilizing p01 results in p02 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 \(\orw\)-token.

For line 3, if b is true, we know p01 or p02 holds before the line. Depending on whether L2 is available or not, the action may or may not consume a level-1 \(\orw\)-token, following the same argument as in line 1 in Fig. 16. If b is false, then p03 holds before the line. Stabilizing this case results in the right branch of the postcondition of line 3, with a \(\orw\)-token for the next round of loop.

It may seem strange that for the loop we do not use a \(Q' \seteq \text{locked}_{\text{cid}}(L1) \lor \text{unlocked}(L1), i.e., the same Q as in Fig. 14. If we use Q' instead, then the case p02 before line 3 cannot have the
while (s=0) dfInc();
and
a

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 \( D, R, G \vdash \{ P \} \Pi : \Gamma \), then

1. both \( \Pi \preceq_{P} \Gamma \) and deadlock-free-\( P \) (\( \Pi \)) hold; and
2. if \( R \Rightarrow \{ R \} \Gamma \) and \( G \Rightarrow \{ G \} \Gamma \), then starvation-free-\( P \) (\( \Pi \)) holds.

Here \( \Pi \preceq_{P} \Gamma \) describes linearizability of the object \( \Pi \). Informally it says that \( \Pi \) has the same effects as the atomic operations of \( \Gamma \) (assuming the initial object states satisfy \( P \)). The formal definition is standard \cite{13} and omitted here. deadlock-free-\( P \) (\( \Pi \)) and starvation-free-\( P \) (\( \Pi \)) are the two progress properties defined following their informal meanings \cite{14} (see Sec. 1).

Theorem 2 shows that LiLi ensures linearizability and deadlock-free behavior together, and it also ensures starvation-freedom when the rely/guarantee specification of the object satisfies certain constraints. The constraints \( R \Rightarrow \{ R \} \Gamma \) and \( G \Rightarrow \{ G \} \Gamma \) 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 define 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 \( dfInc \), the first program must terminate too under fair scheduling.

We propose a novel “progress-aware” object specification \( wr_1(\Gamma) \) for deadlock-free objects that are linearizable with respect to \( \Gamma \). As defined below, \( wr_1(\Gamma) \) wraps the atomic operations in \( \Gamma \) with some auxiliary code for synchronization.

\[ wr_1(\Gamma) = \{ (P, (C); return E) \mid \Gamma(f) = (x, C); return E \} \]

\[ wr_1(C) \triangleq \text{local } u_1 := \text{nondet}(), u_2 := \text{nondet}(); \]

\[ \text{while } (u_1 = 0) \{ \text{lock } l; \text{unlock } l; u_1--; \} \]

\[ (C); \]

\[ \text{while } (u_2 > 0) \{ \text{lock } l; \text{unlock } l; u_2--; \} \]

Here we assume \( 1 \) is a fresh variable, i.e., \( 1 \not\in \{ \Pi, \Gamma, P \} \). The wrapper function \( wr_1 \) inserts a finite (but arbitrary) number of lock-acquire \( (\text{lock } l) \) and lock-release \( (\text{unlock } l) \) actions before and after the atomic abstract code \( C \). The command \( u_1 := \text{nondet}() \) assigns to \( u_1 \) a nondeterministic number. The lock \( l \) is a TAS lock (implementation shown in Fig. 1(b)). Then the progress of a thread executing \( wr_1(\Gamma) \) 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 \( wr_1(\Gamma) \), 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).** \( \Pi \sqsubseteq P \Pi' \) iff \( \forall n, C_1, \ldots, C_n, \sigma_e, \Sigma_n, \sigma_0, \Sigma_0, \{(\sigma_e, \Sigma_e) \Rightarrow P \} \Rightarrow O_{\omega}[\{W, (\sigma_e, \sigma_0)\}] \subseteq O_{\omega}[\{W', (\sigma_e, \Sigma_0)\}] \),

where \( W = \text{let } \Pi \text{ in } C_1 \ldots \ldots C_n \) and \( W' = \text{let } \Pi' \text{ in } C_1 \ldots \ldots C_n \).

Here \( O_{\omega}[W, S] \) generates the full traces of externally observable events in fair executions starting from \( (W, S) \). In our language in Sec. 3 only outputs (produced by \text{print} commands) and fault events are externally observable. Note that \( O_{\omega}[W, S] \) 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 \( 1 \not\in \{ \Pi, \Gamma, P, D, R, G \} \). If \( D, R, G \vdash \{ P \} \Pi : \Gamma \), then

1. \( \Pi \sqsubseteq P \Pi' \) \( \vdash \Gamma(1) \) holds; and
2. if \( R \Rightarrow \{ R \} \Gamma \) and \( G \Rightarrow \{ G \} \Gamma \), then \( \Pi \sqsubseteq P \Gamma \) holds.

where \( wr_1(P) \) extends the precondition \( P \) with the lock variable \( 1 \), i.e., \( wr_1(P) \triangleq (P \land (1 \Rightarrow 0)) \).

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 \( \Pi \) by some more abstract code. For starvation-free objects (case (2) in the theorem), the substitute is \( \Gamma \), the atomic abstract operations. For deadlock-free objects (case (1) in the theorem), the substitute is \( wr_1(\Gamma) \), where the atomic abstract operations \( \Gamma \) are wrapped with auxiliary code for synchronization.

In this paper we do not discuss the verification of clients.

**More details about proofs.** Due to space limit, we omit definitions of some key concepts, e.g., linearizability, deadlock-freedom and starvation-freedom. They are mostly standard and are presented in Appendix B.5. The proofs of Theorems 2 and 4 are shown in detail in Appendix B. Here we only give a brief overview about the structure of the proofs.

To prove Theorem 2 we introduce a termination-preserving simulation, which extends previous work \cite{23} to reason about blocking and delay. LiLi ensures that the concrete implementation \( \Pi \) is simulated by the abstract specification \( \Gamma \) for starvation-free objects and \( wr_1(\Gamma) \) 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 \cite{12}.

Then Theorem 4 follows from these equivalence results and Theorem 2.

In more detail, Theorem 3 is proved in a unified framework in Fig. 18. In the diagram, we use “⇒⇒” for implications and “⇐⇐” for equivalences. The dashed arrow \( a \ldots \ldots b \) and the dotted arrow \( b \ldots \ldots a \) both mean that \( b \) is an instance of \( a \).

The top at Fig. 18 shows our final goals: verifying linearizability \( \Pi \preceq_{P} \Gamma \), and starvation-free-\( P \) (\( \Pi \)) or deadlock-free-\( P \) (\( \Pi \)). Inspired by earlier work \cite{8} \cite{22}, we reduce the goals (see \( \Box \) and \( \lozenge \) in Fig. 18) to verifying contextual refinements \( \sqsubseteq \) in Definition 3.
Π ≼_p \Gamma \land \text{starvation-free}_p(\Pi) \quad \Pi ≼_p \Gamma \land \text{deadlock-free}_p(\Pi)

\begin{align*}
\text{(1)} & \quad \Pi \subseteq_p \Gamma \\
\text{(2)} & \quad \Pi \subseteq_{\text{wr}}(p) \text{ wr}_1(\Gamma) \\
\text{(3)} & \quad \Pi \subseteq_{\text{ wr}}(p) \Pi' \\
\text{(4)} & \quad \Pi \subseteq_{\text{ wr}}(p) \Pi' \\
\text{(5)} & \quad \Pi \subseteq_{\text{ wr}}(p) \Pi' \\
\text{(6)} & \quad \Pi \subseteq_{\text{ wr}}(p) \Pi' \\
\end{align*}

Figure 18. A unified framework for logic soundness proofs.

When using \( \subseteq \) to characterize deadlock-free objects \( \Pi \) (see \( \mathfrak{2} \) in Fig. 18), their methods at the abstract side are no longer atomic. We need to use the “progress-aware” object specification \( \text{wr}_1(\Gamma) \).

Our results \( \mathfrak{1} \) and \( \mathfrak{2} \) in Fig. 18 unify starvation-freedom and deadlock-freedom. Both can be characterized by contextual refinement \( \Pi \subseteq_{\text{ wr}}(p) \Pi' \), where \( \Pi' \) may not be atomic. (See \( \mathfrak{3} \) and \( \mathfrak{4} \).

Next we propose the simulation \( \Pi \subseteq_{\text{ wr}}(p) \Pi' \) as a proof technique for the contextual refinement \( \subseteq \). (See the gray box at the center of Fig. 18.) The simulation ensures that executions of \( \Pi \) preserve the behaviors of \( \Pi' \) under fair scheduling. It is adapted from the compositional simulations in earlier work [21, 24].

At the bottom \( \mathfrak{9} \) of Fig. 18, we define semantics for our logic judgment \( \mathcal{D}, R, G \vDash \{ P \} \Pi : \Gamma \). Note that the logic uses the atomic \( \Gamma \) as specification. The judgment semantics \( \mathcal{D}, R, G \vDash \{ P \} \Pi : \Gamma \) is also based on a simulation, but it is much closer to the logic rules, which can simplify the task of proving the validity of each rule in Fig. 9. It can be directly (see \( \mathfrak{5} \)) translated to \( \Pi \subseteq_{\text{wr}}(p) \text{ wr}_1(\Gamma) \), where \( \text{wr}_1 \) is the wrapper function we just defined (still assume \( \mathfrak{1} \) is an arbitrary fresh variable). If the rely/guarantee conditions \( R \) and \( G \) specify actions of level 0 only (see case (2) in Theorem 2), the judgment semantics can also be reduced to \( \Pi \subseteq \mathcal{D} - \text{gt} \) (see \( \mathfrak{6} \)). Both \( \Pi \subseteq_{\text{wr}}(p) \text{ wr}_1(\Gamma) \) and \( \Pi \subseteq_{\text{ wr}}(p) \Pi' \Pi' \Gamma \) are instances of the more general simulation \( \Pi \subseteq_{\text{ wr}}(p) \Pi' \Pi' \Gamma \) (see \( \mathfrak{5} \) and \( \mathfrak{9} \).

The proof framework in Fig. 18 unifies the logic soundness proofs for both starvation-freedom and deadlock-freedom. All the formal definitions and detailed proofs are given in Appendix B.

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. 19 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.

<table>
<thead>
<tr>
<th>blocking</th>
<th>non-delay</th>
<th>delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-blocking</td>
<td>wait-freedom</td>
<td>lock-freedom</td>
</tr>
<tr>
<td>starvation-freedom</td>
<td>starvation-freedom</td>
<td>deadlock-freedom</td>
</tr>
</tbody>
</table>

Figure 19. Progress properties of concurrent objects.

Our logic LiLi handles blocking by definite actions, and supports delay by \( \mathfrak{8} \)-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. 19.

To verify lock-free objects, we instantiate the definite actions \( \mathcal{D} \) to be false \( \sim \) 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. 9 is reduced to requiring that the loop terminates (the \( \mathfrak{8} \)-tokens decrease at each iteration) unless being delayed by the environment. The definite progress condition \( J \Rightarrow (R, G; \mathcal{D} \mathfrak{8} Q) \) 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 false \( \sim \) true, we also require \( R \) and \( G \) to specify actions of level 0 only, as in Theorem 2.3. 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 [23] 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 Appendix C (for starvation-freex examples) and Appendix D (for deadlock-free examples).

- **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, 24], 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 [25], 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. 4(a) of a mutable set data structure. The algorithm has operations \( \text{add} \), which adds an element to the set, and \( \text{rmv} \), which removes an element from the set. Fig. 20 shows the code and the proof outline for \( \text{rmv} \).

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\[10], 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, acquisitions of a lock will delay other threads competing for the same lock. Thus lock acquirers 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 acquisitions. In our rely and guarantee specifications, the Add and Rmv actions are level-2 delaying actions, while lock acquirers are at level 1.

Next we define the definite actions \( D \) 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 \( D \) 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 \( D \) that a thread is waiting for. It is easy to check that every definite action \( D \) makes the metric to decrease, and that a thread is unblocked to acquire the lock when the metric becomes 0.

In Fig. 20 the precondition is given \( \lhi (1, 2) \), two-level-1 \( \hhi \) tokens for locking two adjacent nodes, and one level-2 \( \hhi \) token for doing Rmv. We apply the (HIDE-\( \hhi \)) rule and assign one \( \hhi \)-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 \( \hhi \)-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 \( p.\text{next} \) 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 level-1 \( \hhi \)-tokens and one more \( \hhi \)-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 \( \hhi \)-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 RWV(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 Appendix D.4.4.

8. Related Work and Conclusion

Using rely-guarantee style logics to verify liveness properties can date back to work by Stark [27], Stølen [28], Abadi and Lamport [11] and Xu et al. [32]. Among them the most closely related work is the fair termination rule for while loops proposed by Stølen [28], 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. 5. 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. [23] 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. [3] 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 Lei no et al. [26] 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 [13] 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 \( D \). But they still do not support starvation-freedom verification.

```
rmv(int e) {
    \{ (1, 2) \land arem(RWV(e)) \land ... \}
    1 local b := false, p, c, n;
    \{ !b \land (1, 2) \land ... \lor b \land ... \}
    2 while (\{ (1, 2) \land ... \}) {
        3 \{ (p, c) := find(e); /\ a loop of list traversal \}
        \{ valid(p, c) \land (1, 2) \land ... \land invalid(p, c) \land (1, 4) \land ... \land ... \}
        4 lock p; lock c;
        \{ valid(p, c) \land (1, 0) \land ... \land invalid(p, c) \land (1, 2) \land ... \land ... \}
        5 b := validate(p, c); /\ a loop of list traversal
        6 if (\{ (1) \land unlock c; unlock p; \}) {
            \{ b \land invalid(p, c) \land (1, 0) \land ...
            \land ... \land ... \land invalid(p, c) \land (1, 2) \land ...
        7 } \land ... \land ... \land ... \land ...
        \{ valid(p, c) \land (1, 0) \land arem(RWV(e)) \land ...
        8 if (c.data = e) {
            n := c.next;
            \{ valid(p, c, e, n) \land (1, 0) \land arem(RWV(e)) \land ...
            \}
        9 \{ p.\text{next} := n; \}
        10 \{ valid(p, n) \land arem(skip) \land ...
        \}
    11 }
    12 unlock c; unlock p;
    \{ arem(skip) \land ...
    \}
}
```

Figure 20. Proofs for optimistic lists (with auxiliary code in gray).
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. [22] 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 [3] 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**

A. The LRG-Style Full Version of LiLi

The full version of LiLi extends the advanced Rely-Guarantee-based logic LRG [7] to support dynamic allocation and ownership transfer. The top level judgment is now in the form of $D, R, G, I \vdash \{P\} : I$. Here the fence $I$ is used to determine the boundary of the shared memory following LRG [7]. Just like $P$, $I$ is also a relational assertion specifying the consistency relation between the concrete data representation and the abstract value.

A.1 LRG-Style Assertions

\[
\begin{align*}
(\text{RelAssn}) & \quad P, q, I, J ::= \ldots \mid \text{own}(x) \\
(\text{RelAct}) & \quad R, G ::= \ldots \mid G * G
\end{align*}
\]

Following LRG [7], we treat program variables as resources [26] and use `own(x)` for the ownership of the program variable $x$. Also we introduce separating conjunction over actions to locally define shared state updates. $G_1 * G_2$ means that the actions $G_1$ and $G_2$ start from disjoint relational states and the resulting states are also disjoint. Here a level-$k$ transition $(\mathcal{S}, \mathcal{S}')$ can be relaxed to a transition at a level $k' \leq k$. Their semantics is defined below.

\[
(\sigma, \Sigma) \cup (\sigma', \Sigma') \triangleq (\sigma \cup \sigma', \Sigma \cup \Sigma') \text{ where } (s, h) \cup (s', h') \triangleq (s \cup s', h \cup h')
\]

\[
\begin{align*}
& (s, h), (s, h) \vdash \text{own}(x) \quad \text{iff} \quad \text{dom}(s \cup s) = \{x\} \\
& (\mathcal{S}, \mathcal{S}') \vdash G_1 * G_2 \quad \text{iff} \quad \exists \mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_1', \mathcal{S}_2', \mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2 \quad \wedge \mathcal{S}' = \mathcal{S}_1' \cup \mathcal{S}_2' \\
& \quad \wedge ((\mathcal{S}_1, \mathcal{S}_1') \vdash G_1) \quad \wedge ((\mathcal{S}_2, \mathcal{S}_2') \vdash G_2) \\
& \mathcal{L}(\mathcal{S}, \mathcal{S}', G_1 * G_2) \triangleq \min(k \mid \exists \mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_1', \mathcal{S}_2', \mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2 \quad \wedge \mathcal{S}' = \mathcal{S}_1' \cup \mathcal{S}_2' \\
& \quad \wedge k = \max(\mathcal{L}(\mathcal{S}_1, \mathcal{S}_1'), G_1), \mathcal{L}(\mathcal{S}_2, \mathcal{S}_2'), G_2))
\end{align*}
\]

\[
R \Rightarrow R' \quad \text{iff} \quad \forall \mathcal{S}, \mathcal{S}', k. ((\mathcal{S}, \mathcal{S}', k) \vdash R) \implies \exists k' \leq k. (\mathcal{S}, \mathcal{S}', k') \vdash R'
\]

The syntactic sugars $\text{Id}$, $\text{Emp}$ and $\text{True}$ represent arbitrary identity transitions, empty transitions and arbitrary transitions respectively.

\[
\begin{align*}
\text{Emp} \triangleq & \quad \text{emp} \times \text{emp} \quad \text{True} \triangleq \quad \text{true} \times \text{true} \quad \text{Id} \triangleq \quad \text{true}
\end{align*}
\]

\textbf{Fence.} Since we logically split states into local and shared parts as in LRG [2], we need a precise invariant $I$ to uniquely determine the boundary between local and shared resources. We define the fence $I \vdash G$ below, which says that the transition $G$ must be made within the boundary specified by $I$.

\[
I \vdash G \quad \text{iff} \quad (I \Rightarrow G) \land (G \Rightarrow (I \Rightarrow I)) \land \text{Precise}(I)
\]

The formal definition of the precise requirement Precise$(I)$ is given in earlier work [21,23], which follows its usual meaning as in separation logic but is now interpreted over relational states. Since the need of fenced rely/guarantee conditions is inherited from LRG and is orthogonal to the problem we study in this paper, readers unfamiliar with LRG can safely ignore it.

A.2 Inference Rules

Figure [21] presents the complete set of inference rules of LiLi.

\textbf{Executing abstract code at (ATOM) and (CSQ) rules.} Below we first define $p \overset{\text{ATOM}}{\Rightarrow} q$ used in the (CSQ) rule.

\[
p \overset{\text{ATOM}}{\Rightarrow} q \quad \text{iff} \quad \forall t, \sigma, \Sigma, u, w, C, \Sigma_F.
\]

\[
(((\sigma, \Sigma), (u, w), C) \vdash p) \land (\Sigma \cup \Sigma_F) \implies \exists k, u', w', C', \Sigma'.
\]

\[
((C, \Sigma \cup \Sigma_F) \Rightarrow \lambda_{\Sigma_F} (C', \Sigma' \cup \Sigma_F)) \land ((\sigma, \Sigma), (\sigma, \Sigma), k) \vdash G \ast \text{True}
\]

\[
\land ((\sigma, \Sigma'), (u', w'), C') \vdash q \land (u', w') <_k (u, w) \quad (<_k \text{ defined in Fig. [12]})
\]

The definition of $p \overset{\text{ATOM}}{\Rightarrow} q$ is similar to $p \Rightarrow q$ defined in Fig. [13]. In addition to executing the abstract code and decreasing the corresponding black tokens, $p \overset{\text{ATOM}}{\Rightarrow} q$ also requires the overall transition to satisfy $G$. 


for all \( f \in \text{dom}(\Pi) \):
\[
\Pi(f) = (x, C) \quad \Gamma(f) = (y, C) \quad \text{dom}(\Pi) = \text{dom}(\Gamma)
\]

\[
\forall \Gamma, \Phi, \Phi' \vdash \{ P \circ \text{own}(x) \} \wedge (x = y) \wedge \text{arem}(C) \wedge \Phi \}
\]

\[
D, R, G, I \vdash \{ P \circ \text{own}(x) \} \wedge (x = y) \wedge \text{arem}(C) \wedge \Phi \} (\text{FRM})
\]

\[
D, R, G, I \vdash \{ P \circ \text{own}(x) \} \wedge (x = y) \wedge \text{arem}(C) \wedge \Phi \} \quad (\text{AREM})
\]

\[
D, R, G, I \vdash \{ p \circ \text{own}(x) \} \wedge (x = y) \wedge \text{arem}(C) \wedge \Phi \} \quad (\text{DISI})
\]

**Figure 21.** LRG-style inference rules.
B. Logic Soundness Proofs

In this appendix, we give the detailed proofs of Theorem 2 following the unified proof framework in Fig. 18.

- Appendix B.1 defines the judgment semantics $\mathcal{D}, R, G \models \{P\} \Pi : \Gamma$.
- Appendix B.2 shows the proofs of $\Theta$ of Fig. 18 i.e., the logic rules are sound with respect to the judgment semantics.
- Appendix B.3 defines the “common simulation” $\Pi \preceq_{\mathcal{P}} \Pi'$. Appendix B.3.1 shows the proofs of $\Theta$ and $\Theta$ of Fig. 18 i.e., the judgment semantics $\mathcal{D}, R, G \models \{P\} : \Gamma$ implies instantiations of the common simulation.
- Appendix B.4 shows the core proofs for the gray box at the center of Fig. 18 i.e., the common simulation implies $\Pi \subseteq_{\mathcal{P}} \Pi'$, the contextual refinement under fair scheduling.
- Appendix B.5 shows the proofs of $\Theta$ and $\Theta$ of Fig. 18 i.e., deadlock-freedom/starvation-freedom and linearizability can be characterized by the contextual refinements.

B.1 Judgment Semantics

The judgment semantics $\mathcal{D}, R, G \models \{P\} : \Gamma$ is based on a simulation between $\Pi$ and $\Gamma$, with four well-founded metrics: $M, \xi, M'$ and $u$. The metric $M$ is to ensure that the current thread $t$ must fulfill its definite action $D_t$ in a finite number of steps. If thread $t$ is blocked, the metric $\xi$ specifies the set of the environment threads that $t$ is waiting for. It shrinks when an environment thread $t'$ finishes definite action $D_{t'}$. The metric $M'$ corresponds to the number of white tokens $\xi$, which is to ensure that thread $t$ progresses on its own when $\xi$ becomes empty. The last metric $u$ corresponds to the tuple of black tokens. It bounds the number of actions made by thread $t$ which could delay the progress of its environment threads.

**Definition 5.** $\mathcal{D}, R, G \models \{P\} : \Gamma$ iff, for any $f \in \text{dom}(\Pi)$, for any $\sigma$ and $\Sigma$, for any $t$, if $\Pi(f) = (x, C)$, $\Gamma(f) = (y, C)$ and $(\sigma, \Sigma) \models t_{\Pi} \star \text{own}(x) \star \text{own}(y) \land (x = y)$, there exist four well-founded metrics $u, M, M'$ and $\xi$ and a set $\xi \in \mathcal{P}(\text{ThrId})$ such that $\mathcal{D}, R, G \models \{C, \sigma\} \preceq (C, \Sigma) \circ (u, M, M') \downarrow_{\xi} (P \star \text{own}(x) \star \text{own}(y))$.

Here $\mathcal{D}, R, G \models \{C, \sigma\} \preceq (C, \Sigma) \circ (u, M, M') \downarrow_{\xi} Q$ is co-inductively defined as follows. Whenever $\mathcal{D}, R, G \models \{C, \sigma\} \preceq (C, \Sigma) \circ (u, M, M') \downarrow_{\xi} Q$ holds, then the following hold:

1. Suppose $\sigma = (s, h)$. Then $\xi \subseteq s(\text{TIDS})$ and $t \notin \xi$.
   For any $t' \in \xi$, we have $(\sigma, \Sigma) \models \text{Enabled}(D_{t'}) \models \text{true}.$
2. If $C = E_1 \text{ return } E_2$, then for any $\Sigma_F$ such that $\Sigma \subseteq \Sigma_F$, there exist $E$ and $\Sigma'$ such that (a) $(C, \Sigma \circ E_1) \rightarrow_{\Sigma\Sigma}^* ((E, \Sigma') \circ \text{return} E_2)$, and (b) $(\sigma, \Sigma) \models E_1, \Sigma$ and (c) $((\sigma, \Sigma), (\sigma, \Sigma'), 0) \models \text{true}.$
3. For any $\sigma_F$, $(C, \sigma \circ \sigma_F) \rightarrow_{\Sigma} \text{ abort}$.
4. For any $C', \sigma''$, $\sigma_F$ and $\Sigma_F$, if $(C, \sigma \circ \sigma_F) \rightarrow_{\Sigma\Sigma}^* (C', \sigma'')$ and $\Sigma \subseteq \Sigma_F$, then there exist $\sigma', \Sigma', k, u', M', M'$ and $\xi'$ such that (a) $\sigma'' = \sigma' \circ \sigma_F$, and (b) $(C, \Sigma \circ E_1) \rightarrow_{\Sigma\Sigma}^* ((C', \Sigma') \circ \text{return} E_2)$, and (c) $(\sigma, \Sigma) \models E_1, \Sigma$ and (d) $((\sigma, \Sigma), (\sigma', \Sigma'), k) \models \text{true}.$
5. If $f(k) = (x, C, \sigma') \preceq (C, \Sigma') \circ (u', M', M') \downarrow_{\xi'} Q$ and (a) $f(k) = (x, C, \sigma') \preceq (C, \Sigma') \circ (u', M', M') \downarrow_{\xi'} Q$ and (b) $\xi = \{t' \in \xi \land ((\sigma, \Sigma), (\sigma', \Sigma')) \models (D_{t'}) \models \text{true}.$
   Then $\xi' \preceq \xi$ and $u' \approx_k u$ and (c) if $k = 0$, then either $M' < M$, or $M' = M$ and $\xi = \emptyset$ and (d) if $k = 0$ and $(\sigma, \Sigma) \models \text{true}.$

**Definition 6.** $\mathcal{D}, R, G \models \{P\} C[q]$ iff, for any $\sigma, \Sigma, u, w, \Sigma$, for any $t$, if $(\sigma, \Sigma, (u, w, \Sigma)) \models \text{true}$, then $\mathcal{D}, R, G \models \{C, \sigma\} \preceq (C, \Sigma) \circ (u, w, w, \Sigma) \downarrow_{\xi} q$.

Here $\mathcal{D}, R, G \models \{C, \sigma\} \preceq (C, \Sigma) \circ (u, w, w, w, \Sigma) \downarrow_{\xi} q$ is co-inductively defined as follows. Whenever $\mathcal{D}, R, G \models \{C, \sigma\} \preceq (C, \Sigma) \circ (u, w, w, w, \Sigma) \downarrow_{\xi} q$ holds, then the following hold:

1. Suppose $\sigma = (s, h)$. Then $\xi \subseteq s(\text{TIDS})$ and $t \notin \xi$.
   For any $t' \in \xi$, we have $(\sigma, \Sigma) \models \text{Enabled}(D_{t'}) \models \text{true}.$
   If $\xi = \emptyset$, then $|\Sigma| > 1$.
2. If $C = \text{ skip}$, then for any $C_F$ such that $\Sigma \subseteq \Sigma_F$, there exist $\Sigma' \preceq \Sigma'$ such that (a) $(C, \Sigma \circ E_1) \rightarrow_{\Sigma\Sigma}^* (C', \Sigma' \circ E_2)$, and (b) $(\sigma, \Sigma') \models E_1, \Sigma'$ and (c) $\Sigma' = ((0, 0), 0)$ and $\Sigma = (0, 0)$ and $\xi = \emptyset$.
   Then $\Sigma' \preceq \Sigma'$ and $\xi' \preceq \xi'$ and $u' \approx_k u'$ and (d) $((\sigma, \Sigma), (\sigma', \Sigma') \models \text{true}.$
If $C = E[\text{return } E]$, then for any $\Sigma_F$ such that $\Sigma \cup \Sigma_F$, there exist $\Sigma$ and $\Sigma'$ such that

(a) $(C', \Sigma', \Sigma_F) \rightarrow^*_t (\text{return } E, \Sigma', \Sigma_F)$, and

(b) $(\langle \sigma, \Sigma', (u, w), \text{skip} \rangle \models q_i$ and $[E]_{\sigma, u} = [E]_{\Sigma', u}$, and

(c) $(\langle \sigma, \Sigma, (\sigma, \Sigma') \rangle, 0) \models G_t \land \text{True}.

For any $\sigma, \sigma' \in \Sigma_F$, if $(C, \sigma \cup \sigma F) \rightarrow_t (C', \sigma')$, then there exist $\sigma', \sigma', \Sigma', k, u', w', \text{ws}', \text{ws}'$ and $\xi'$ such that

(a) $\sigma'' = \sigma' \cup \sigma_F$, and

(b) $(\langle C', \sigma' \rangle, \langle \sigma', \Sigma' \rangle) \rightarrow^*_t (\langle C', \sigma' \rangle, \langle \sigma', \Sigma' \rangle)$, and

(c) $(\langle \sigma, \Sigma, (\sigma', \Sigma') \rangle, k) \models G_t \land \text{True}$, and

(d) $(\langle \sigma, \Sigma, (\sigma', \Sigma') \rangle, k) \models G_t \land \text{True}$, and

(e) either $u' < k$ or $u' = k = 0$ and $w' <_H \text{ws}'$ and $\text{ws}' = w$, or $u' = k = 0$ and $w' = \text{ws}'$ and $w' = w$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$; and

(f) if $(\langle \sigma, \Sigma, (\sigma', \Sigma') \rangle) \models (\langle D_t \rangle) \land \text{True}$, then $\text{ws}' <_H \text{ws}$.

Below we also define the semantics for the sequential judgment used in the ATOM rule. Note that $C$ only accesses the concrete memory $\sigma$, therefore we require the other components in the full state (i.e., $\omega$, $w$, $C$ and $\Sigma$) should remain unchanged during the execution of $C$.

Definition 7 (SL judgment semantics, total correctness). $\models [p]C[q]$ iff, for all $\sigma, \Sigma, u, w$ and $C$, for any $t$, if $(\langle \sigma, \Sigma, (u, w), C \rangle) \models p$, the following hold:

1. for any $\sigma'$, if $(C, \sigma) \rightarrow^*_t (\text{skip}, \sigma')$, then $(\langle \sigma', \Sigma, (u, w), C \rangle) \models q_i$;

2. $(C, \sigma) \rightarrow^*_t \text{abort}$;

3. $(C, \sigma) \rightarrow^*_t$.

Lemma 8. If $\vdash [p]C[q]$, then $\models [p]C[q]$.

Definition 9 (Locality). Locality $(C)$ iff, for any $\sigma_1$ and $\sigma_2$, let $\sigma = \sigma_1 \cup \sigma_2$, then the following hold:

1. (Safety monotonocity) If $\langle C, \sigma, 1 \rangle \rightarrow^*_t \text{abort}$, then $(C, \sigma) \rightarrow^*_t \text{abort}$.

2. (Termination monotonicity) If $\langle C, \sigma_1 \rangle \rightarrow^*_t \text{abort}$ and $\langle C, \sigma_1 \rangle \rightarrow^*_t \text{abort}$, then $(C, \sigma) \rightarrow^*_t \text{abort}$.

3. (Frame property) For any $n$ and $\sigma'$, if $(C, \sigma_1) \rightarrow^*_n \text{abort}$ and $(C, \sigma) \rightarrow^*_n (C', \sigma')$, then there exists $\sigma_1'$ such that $\sigma' = \sigma_1' \cup \sigma_2$ and $(C, \sigma_1) \rightarrow^*_n (C', \sigma_1')$.

B.1.1 Instantiating Metrics and Well-Founded Orders

The judgment semantics in Definition 5 can be viewed as an instantiation of the simulation in Definition 5. The key is to instantiate the metrics $M$ and $M$ in $D, R, G \models i (C, \sigma) \rightarrow^*_n (C', \sigma')$ or $(u, w, M) \models H$. Below we take the instantiation for $M$ as an example.

\[
\begin{array}{l}
\text{(Metric)} \quad M \quad := \quad (ws, H) \\
\text{(W/Stack)} \quad ws \quad := \quad (w, n) \mid (w, n) :: ws \\
\text{(StkHeight)} \quad H \quad \in \quad \text{Nat}
\end{array}
\]

For each single thread, its metric $ws$ is usually a list of $(w, n)$ pairs, where $w$ is the while-specific metric (which is left to be instantiated later) and $n$ is a natural number specifying the “code size” as in Liang et al.’s work [23]. We let the threaded metric $ws$ be a list (a stack actually) to allow different while-specific metrics for nested loops. That is, when entering a loop, we can push a $(w, n)$ pair to the $ws$ stack; and when exiting the loop, we pop the pair out of $ws$.

The threaded metric $ws$ follows the dictionary order. However, the usual dictionary order over lists is not well-founded (consider $B > AB > AAB > AAAAB > \ldots$ in a dictionary). To address this issue, we introduce a bound of the list length (stack height), $H$, and define the well-founded order $\prec_H$ by requiring the length of the lists should not be larger than $H$. Intuitively, the stack height $H$ represents the maximal depth of nested loops, so it can be determined for any given program. The well-founded orders $M' < M$ and $ws' \prec_H ws$ are defined as follows.

\[
\begin{align*}
\text{ws'} \prec_H \text{ws} & \quad \text{iff} \quad (\text{ws'} \prec \text{ws}) \land (|\text{ws'}| \leq H) \land (|\text{ws}| \leq H) \\
\text{ws'} \prec_H \text{ws} & \quad \text{iff} \quad (\text{ws'} \prec_H \text{ws}) \land (|\text{ws'}| = |\text{ws}|)
\end{align*}
\]
\[(w', n') < (w, n)\]  \[\iff (w' < w) \lor (w' = w \land n' < n)\]
\[(w', n') = (w, n)\]  \[\iff (w' = w) \land (n' = n)\]
\[(w', n') \leq (w, n)\]  \[\iff (w', n') < (w, n) \lor (w', n') = (w, n)\]
\[(w', n') = (w, n)\]  \[\iff (w', n') \leq (w, n) \lor (w', n') = (w, n)\]

The well-founded order over the \((w, n)\) pairs is a usual dictionary order below, where the order over \(w\) is instantiated later depending on the type of \(w\).

\[(w', n') < (w, n)\]  \[\iff (w' < w) \lor (w' = w \land n' < n)\]

\[(w', n') = (w, n)\]  \[\iff (w' = w) \land (n' = n)\]
\[(w', n') \leq (w, n)\]  \[\iff (w', n') < (w, n) \lor (w', n') = (w, n)\]

**Lemma 10 (Well-foundedness).** The relations \(M' < M\) and \(ws' <_H ws\) defined above are both well-founded relations.

The judgment semantics in Definition 6 has more restrictions on the well-founded order over \(ws\). We define the order \(ws' <_H ws\) below, which is stronger than the more general order \(ws' <_H ws\) above.

\[ws' <_H ws\]  \[\iff (ws' \ll ws) \land (|ws'| \leq \mathcal{H}) \land (|ws| \leq \mathcal{H})\]
\[ws' \leq_H ws\]  \[\iff (ws' \ll_H ws) \lor (ws' = ws)\]
\[(w', n') < (w, n)\]  \[\iff (w', n') < (w, n) \lor (w', n') = (w, n)\]
\[(w', n') \leq (w, n)\]  \[\iff (w', n') < (w, n) \lor (w', n') = (w, n)\]
\[(w', n') = (w, n)\]  \[\iff (w', n') \leq (w, n) \lor (w', n') = (w, n)\]
\[(w', n') \ll (w, n)\]  \[\iff (w', n') < (w, n) \lor (w', n') = (w, n)\]

**Height \(\mathcal{H}\).** As we said, the stack height \(\mathcal{H}\) represents the maximal depth of nested loops. For any given program \(C\), we can determine the stack height using a function \(\text{height}\) defined below.

\[\text{height}(\text{skip}) \overset{\text{def}}{=} 1\]
\[\text{height}(\text{return } E) \overset{\text{def}}{=} 1\]
\[\text{height}(c) \overset{\text{def}}{=} 1\]
\[\text{height}(C) \overset{\text{def}}{=} 1\]
\[\text{height}(C_1; C_2) \overset{\text{def}}{=} \max\{\text{height}(C_1), \text{height}(C_2)\}\]
\[\text{height}(\text{if } (B) C_1 \text{ else } C_2) \overset{\text{def}}{=} \max\{\text{height}(C_1), \text{height}(C_2)\}\]
\[\text{height}(\text{while } (B) \{C\}) \overset{\text{def}}{=} \text{height}(C) + 1\]

**Initial code size.** In Definition 6 the judgment semantics initially takes the static code size \(|C|\) defined below as the second dimension of \(ws\) and \(ws\).

\[|\text{skip}| \overset{\text{def}}{=} 0\]
\[|\text{return } E| \overset{\text{def}}{=} 1\]
\[|c| \overset{\text{def}}{=} 1\]
\[|C| \overset{\text{def}}{=} 1\]
\[|C_1; C_2| \overset{\text{def}}{=} |C_1| + |C_2| + 1\]
\[|\text{if } (B) C_1 \text{ else } C_2| \overset{\text{def}}{=} \max\{|C_1|, |C_2|\} + 1\]
\[|\text{while } (B) \{C\}| \overset{\text{def}}{=} 1\]

**Example of \(ws\).** Below we use a simple example to show how we assign a proper \(ws\) to each state during an execution. In the code below, we assign different labels to different layers of a nested while loop. The initial code is \(\text{while}(i > 0) \ i--;\) The loop is labeled with 1 and its body code is labeled with 2. After the loop is unfolded, we use the syntax while to be distinguished from the original while which has not been unfolded.
In the $ws$ below, the first dimension specifies the number of iterations left to unfold, and the second dimension specifies the "code size" at each layer.

<table>
<thead>
<tr>
<th>$C$</th>
<th>$\sigma$</th>
<th>$ws$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \rightarrow$</td>
<td>$\iota = 2$</td>
<td>$(0,1)$</td>
</tr>
<tr>
<td>while$^1(\iota &gt; 0) \ i\leftarrow^2$;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2 \rightarrow$</td>
<td>$\iota = 2$</td>
<td>$(0,0):(1,2)$</td>
</tr>
<tr>
<td>while$^1(\iota &gt; 0) \ i\leftarrow^2$;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$3 \rightarrow$</td>
<td>$\iota = 1$</td>
<td>$(0,0):(1,1)$</td>
</tr>
<tr>
<td>$\text{skip}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$4 \rightarrow$</td>
<td>$\iota = 1$</td>
<td>$(0,0):(1,0)$</td>
</tr>
<tr>
<td>while$^1(\iota &gt; 0) \ i\leftarrow^2$;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$5 \rightarrow$</td>
<td>$\iota = 1$</td>
<td>$(0,0):(0,2)$</td>
</tr>
<tr>
<td>while$^1(\iota &gt; 0) \ i\leftarrow^2$;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$6 \rightarrow$</td>
<td>$\iota = 0$</td>
<td>$(0,0):(0,1)$</td>
</tr>
<tr>
<td>$\text{skip}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$7 \rightarrow$</td>
<td>$\iota = 0$</td>
<td>$(0,0):(0,0)$</td>
</tr>
<tr>
<td>while$^1(\iota &gt; 0) \ i\leftarrow^2$;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$8 \rightarrow$</td>
<td>$\iota = 0$</td>
<td>$(0,0)$</td>
</tr>
<tr>
<td>$\text{skip}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Initially, $ws$ is $(0,1)$; the first dimension is 0 because we have not started to unfold the loop, and the second dimension is 1 because the code size of the whole loop is 1. After one step of the loop, $ws$ becomes $(0,0):(1,2)$. Since we have unfolded the loop, the $ws$ stack contains two pairs now. In the second pair, the first dimension is 1 because the loop needs only one more iteration to finish (i.e., we only need to unfold it one more time). Its second dimension is 2 because the size of the loop body code is 2. After the next step, this dimension decreases. At the step of line 5, we unfold the loop again. So the first dimension of the second pair decreases to 0, saying that we do not need to unfold the loop anymore. Finally, at the step of line 8, the loop finishes, thus we pop out the second pair of the $ws$ stack.
B.2 Soundness of the Inference Rules

In this section, we prove Lemma \([\text{L12}]\) by induction over the derivation.

**Lemma 11** (\([\text{L11}]\) in Fig. [B.5]). If \(D, R, G, I \vdash \{P\} \Pi : \Gamma\), then \(D, R, G \models \{P\} \Pi : \Gamma\).

**Proof.** By induction over derivation. By Lemma \([\text{L12}]\) we only need to prove the following:

\[
D, R, G \models \{P \ast \text{own}(x) \ast \text{own}(y) \land (x = y) \land \text{arem}(C) \land \text{arem}(\text{skip})\}
\]

then \(D, R, G \models \{P\} \Pi : \Gamma\).

(B.1)

By co-induction.

**Lemma 12.** If

1. \(D, R, G, I \vdash \{p\} C\{q\}\);
2. \(\text{Enabled}(D) \Rightarrow I\);
3. for any \(t\) and \(t'\) such that \(t \neq t'\), we have \(G_t \Rightarrow R_{t'}\);

then \(D, R, G \models \{p\} C\{q\}\).

In the following subsections, we prove Lemma \([\text{L12}]\) by induction over the derivation.

B.2.1 The **WHL Rule**

**Lemma 13** (**WHL-Sound**). If

1. \(p \land B \Rightarrow p' ; p \land B \land Q \ast \text{true} \Rightarrow p' \ast (\circ \land \text{emp})\);
2. \(D, R, G : \models \{p'\} C\{p\}\);
3. \(p \Rightarrow (B = B) \ast J ; J \Rightarrow I ; \text{Sta}(J, R \lor G) ; Q \Rightarrow I\);
4. \(J \Rightarrow (R, G ; D \downarrow J ; Q) ; D' \leq D ; \text{wffAct}(R, D')\);
5. \(I \Rightarrow \{R, G\} ; \text{Sta}(p, R \ast Id) ; \text{Enabled}(D) \Rightarrow I\);
6. for any \(t\) and \(t'\) such that \(t \neq t'\), we have \(G_t \Rightarrow R_{t'}\);

then \(D, R, G \models \{p\} \text{while} (B)\{C\}\{p \land \neg B\}\).

**Proof.** Let \(H = \text{height}(\text{while} (B)\{C\}) = \text{height}(C) + 1\). We know \(|\text{while} (B)\{C\}| = 1\). Let

\[ws = ((0, 0), 1)\quad \text{and} \quad ws = (0, 1)\quad \text{and} \quad \xi = \emptyset\]

Below we prove: for any \(t\), for any \(\sigma, \Sigma, u, w\) and \(C\), if \((\langle \sigma, \Sigma \rangle, (u, w), C) \models p_t\), then

\[
D, R, G \models \{p\} \text{while} (B)\{C\}\{\sigma, \Sigma\} \models (C, \Sigma) \circ (u, ws, ws, w, H) \Downarrow_{p} p \land \neg B .
\]

By co-induction. Suppose \(\sigma = (s, h)\). Since \(p \Rightarrow (B = B) \ast I\), we know

\[
(\sigma, \Sigma) \models I \ast \text{true} , \quad \text{and either } [B]_s = \text{true} \text{ or } [B]_s = \text{false} .
\]

Since \(p \Rightarrow J \ast \text{true} , \text{we know}

\[
(\sigma, \Sigma) \models J \ast \text{true} .
\]

We only need to prove the following (1)(2)(3)(4)(5)(6).

(1) For any \(t \in \xi\), we have \((\sigma, \Sigma) \models \text{Enabled}(D_t) \ast \text{true} .

**Proof.** It is vacantly true.

(2) If \(\text{while} (B)\{C\} = \text{skip} \), then ....

**Proof.** It is vacantly true.

(3) If \(\text{while} (B)\{C\} = E \ast \text{return} E\), then ....

**Proof.** It is vacantly true.

(4) For any \(\sigma, \Sigma, (\langle \sigma, \Sigma \rangle, (u, w), C) \models p_t\), then \(\text{with} (B)\{C\}, (u, w, \Sigma) \models \text{true} .

**Proof.** It holds because \([B]_s\) is not undefined.

(5) If \(\text{while} (B)\{C\}, (u, w, \Sigma) \models \text{true} .

**Proof.** It holds because \([B]_s\) is not undefined.

(6) If \(\text{while} (B)\{C\}, (u, w, \Sigma) \models \text{true} .

**Proof.** It holds because \([B]_s\) is not undefined.
(f) if $((σ, Σ), (σ', Σ')) \models (D, i)$ * True, then $ws' <_H ws$.

Proof: We have two cases depending on whether $[B]_o$ is *true* or *false*.

(I) If $[B]_o = \text{true}$, we know $σ'' = σ \cup σ_F$ and $(\text{while } (B)(C'), σ \cup σ_F) \rightarrow_t (C; \text{while } (B)(C), σ \cup σ_F)$.

Also we know $((σ, Σ), (u, w, Σ)) \models p_k ∧ B$. From $p_k ∧ B ⇒ p'$ and $D, R, G \models \{ p' \} C(p)$, let $w_s = (0, 0), |C|)$ and $w_s = (0, |C|)$, we get:

$$D, R, G \models (C, σ) ≤ (C, Σ) ∘ (u, ws, ws', w, Σ), \text{height}(C) \{p\}.$$

Let

$$ws' = (0, 0) :: (w, |C'| + 1),$$

we know $ws' <_H ws$. Let

$$k_s = f(σ, Σ) \text{ and } ξ_0 = \{ t'' | (t'' \neq t) \land ((σ, Σ) \models \text{Enabled}(D_0^r) \ast \text{true}) \}.$$

Since $J ⇒ (R, G; D' \rightarrow Q)$, we have two cases:

• $((σ, Σ) \models \exists t'. \text{Enabled}(D_0^r) \ast \text{true})$.

Then we know $ξ_0 \neq \emptyset$. By Lemma 14 let

$$ws' = (0, 0) :: ((k_s, w), 0).$$

Also we know $ws' <_H ws$.

• $((σ, Σ) \models Q_i \ast \text{true})$.

By Lemma 14 let

$$ws = (0, 0) :: ((k_s, w), |C| + 1).$$

we know

$$D, R, G \models (C; \text{while } (B)(C), σ) ≤ (C, Σ) ∘ (u, ws, ws', w, Σ), \text{height}(C) \{p\} \{p\} ∧ \neg B.$$

Also, $ws' <_H ws$ holds.

(II) If $[B]_o = \text{false}$, we know $(\text{while } (B)(C), σ \cup σ_F) \rightarrow_t (\text{skip}, σ \cup σ_F)$.

By (skip) rule, let

$$ws' = (0, 0) \text{ and } ws' = (0, 0).$$

we know

$$D, R, G \models (\text{skip}, σ) ≤ (C, Σ) ∘ (u, ws, ws', w, Σ), \text{height}(C) \{p\} \{p\} ∧ \neg B.$$

Also we know $ws' <_H ws$ and $ws' <_H ws$.

Since $((σ, Σ), (σ', Σ'), 0) \models [I] \ast \text{true}$, we know

$$((σ, Σ), (σ', Σ'), 0) \models [I] \ast \text{true}. $$

Since $I \models G$, we know

$$((σ, Σ), (σ', Σ'), 0) \models G_i \ast \text{true}. $$

(6) If $((σ, Σ), (σ', Σ'), k) \models R_c \ast \text{Id}$, then there exist $u', ws', ws', w', ξ_d$ and $ξ'$ such that

(a) $D, R, G \models (\text{while } (B)(C), σ') ≤ (C, Σ') ∘ (u', ws', ws', w', Σ), \text{height}(C) \{p\} \{p\} ∧ \neg B,$ and

(b) $ξ_d = \{ t' | (t' \in ξ) \land ((σ, Σ), (σ', Σ')) \models (D_0^c) \ast \text{Id} \}.$

(c) $ξ = 0 ⇒ ξ \subseteq ξ'$ and $u' \approx_k u,$ and

(d) if $k = 0$ and $σ, σ_F$ are $\text{Enabled}(D_0^c) \ast \text{true},$ then $ws' <_H ws$.

Proof: Since Sta($p, R, \text{Id}$), we know there exist $u'$ and $w'$ such that

$$((σ', Σ'), (u', w'), Σ) \models p_k \text{ and } u' \approx_k u \text{ and } k = 0 ⇒ w' = w.$$

By the co-induction hypothesis, we get

$$D, R, G \models (\text{while } (B)(C), σ') ≤ (C, Σ') ∘ (u, ws, ws, w', Σ), \text{height}(C) \{p\} \{p\} ∧ \neg B.$$

If $k = 0$, we know $w' = w$ and $ξ_d = \emptyset$.

Thus we are done. □

We define:

$$\text{head}(ws) \overset{def}{=} \begin{cases} ((n_1, n_2), n_3) & \text{if } ws = ((n_1, n_2), n_3) \\ ((n_1, n_2), n_3) & \text{if } ws = ((n_1, n_2), n_3) \end{cases}$$

$$\text{inhead}(ws, ((k_1, k_2), k_3)) \overset{def}{=} \begin{cases} ((n_1 + k_1, n_2 + k_2, n_3 + k_3) & \text{if } ws = ((n_1, n_2), n_3) \\ ((n_1 + k_1, n_2 + k_2, n_3 + k_3) & \text{if } ws = ((n_1, n_2), n_3) \end{cases}$$

Lemma 14. If

1. $p \land B ⇒ p' ; p \land B, Q \ast \text{true} ⇒ p' \ast (\emptyset \land \text{emp});$
2. $D, R, G \models \{ p' \} C(p);$ $p ⇒ (B = B) \ast J; J ⇒ I; \text{Sta}(J, R \lor G); Q ⇒ I.$

24
4. $J \vdash (R, G; D' \downarrow \to Q); D' \leq D; \text{wffAct}(R, D')$;
5. $I \vdash \{ R, G \}; \text{Sta}(p, R \ast \text{Id}); \text{Enabled}(D) \Rightarrow I$;
6. for any $t$ and $t'$ such that $t \neq t'$, we have $G_t \Rightarrow R_{t'}$;
7. $D, R, G \models \{ C_1, \sigma \} \preceq (C, \Sigma) \circ (u, w_s, w_{s1}, w_1, H) \downarrow \xi, p$;
8. $(\sigma, \Sigma) \models J * \text{true}; \text{height}(C) = H; \text{head}(w_{s1}) = ((n_1, n_2), n_3); f(\sigma, \Sigma, i) = k_i; w_1 \leq w$;
9. $\xi_0 = \{ t' | (t' \neq t) \land (\sigma', \Sigma') \models \text{Enabled}(D'_t) \ast \text{true} \}; \xi = \xi_0 \cup \xi_1$;
10. $w_s = (0, 0) :: \text{inhead}(w_{s1}, (w_1, 1))$;
11. one of the following holds:
   • $(\sigma, \Sigma) \models Q_t \ast \text{true}; w_s = ((0, 0), 0) :: \text{inhead}(w_{s1}, ((k_i, w_1), -n_3))$;
   • $(\sigma, \Sigma) \models Q_t \ast \text{true}; w_s = ((0, 0), 0) :: \text{inhead}(w_{s1}, ((k_i, w_1), 1))$;
then $D, R, G \models \{ C_1 \}; \text{while} (B) \{ C \}, \sigma \preceq (C, \Sigma) \circ (u, w_s, w_{s1}, w_1, H + 1) \downarrow \xi \ p \land \neg B$.

Proof: By co-induction. We only need to prove the following (1)(2)(3)(4)(5).

(1) For any $t' \in \xi$, we have $(\sigma, \Sigma) \models \text{Enabled}(D_{t'}) \ast \text{true}$.
   Proof: From
   
   $D, R, G \models \{ C_1, \sigma \} \preceq (C, \Sigma) \circ (u, w_{s1}, w_{s1}, w_1, H) \downarrow \xi, p$,

   we know for any $t' \in \xi_1$, we have $(\sigma, \Sigma) \models \text{Enabled}(D_{t'}) \ast \text{true}$. Since $D' \leq D$, we know for any $t' \in \xi_0$, we have $(\sigma, \Sigma) \models \text{Enabled}(D_{t'}) \ast \text{true}$. Thus we are done.

(2) If $(C_1; \text{while} (B) \{ C \}) = \text{skip}$, then ...
   Proof: It is vacantly true.

(3) For any $\sigma_F, (C_1; \text{while} (B) \{ C \}, \sigma \uplus \sigma_F) \not\rightarrow t \text{ abort}$.
   Proof: From
   
   $D, R, G \models \{ C_1, \sigma \} \preceq (C, \Sigma) \circ (u, w_{s1}, w_{s1}, w_1, H) \downarrow \xi, p$,

   we know $(C_1, \sigma \uplus \sigma_F) \not\rightarrow t \text{ abort}$. By the operational semantics, we are done.

(4) If $(C_1; \text{while} (B) \{ C \}, \sigma \uplus \sigma_F) \not\rightarrow (C'', \sigma'')$, then there exist $\sigma', C', \Sigma', k, u', w_{s'}, w_{s'}', w', \xi'$ such that
   (a) $\sigma'' = \sigma' \uplus \sigma_F$ and
   (b) $(C, \Sigma \uplus \Sigma_F) \not\rightarrow t' (C', \Sigma' \uplus \Sigma_F)$, and
   (c) $D, R, G \models \{ C', \sigma' \} \preceq (C', \Sigma') \circ (u', w_{s'}, w', H + 1) \downarrow \xi' \ p \land \neg B$; and
   (d) $(\{ (\sigma, \Sigma), (\sigma', \Sigma') \}, k) \models G_t \ast \text{True}, \text{and}$
   (e) either $u' < k, u'$
      or $u' = u$ and $k = 0$ and $w_{s'}' <_{R+1} \ w_s$ and $w' = w$,
   or $u' = u$ and $k = 0$ and $w_{s'} = w_s$ and $w' = w$ and $\xi \neq \emptyset$ and $\xi \subseteq \xi'$; and
   (f) if $(\{ (\sigma, \Sigma), (\sigma', \Sigma') \}) \models \{ (D_1) \} \ast \text{True},$ then $w_{s'}' <_{R+1} w_{s1}$.

   Proof: We have two cases depending on whether $C_1$ is $\text{skip}$ or not.
   (I) If $(C_1; \text{while} (B) \{ C \}, \sigma \uplus \sigma_F) \not\rightarrow (C''; \text{while} (B) \{ C \}, \sigma'')$, then $(C_1, \sigma \uplus \sigma_F) \not\rightarrow (C', \sigma'')$. From $D, R, G \models \{ C_1, \sigma \} \preceq (C, \Sigma) \circ (u, w_{s1}, w_{s1}, w_1, H) \downarrow \xi, p$, we know there exist $\sigma', C', \Sigma', k, u', w_{s'}, w_{s'}', w_1$ and $\xi_1$ such that
   (A) $\sigma'' = \sigma' \uplus \sigma_F$, and
   (B) $(C, \Sigma \uplus \Sigma_F) \not\rightarrow t' (C', \Sigma' \uplus \Sigma_F)$, and
   (C) $D, R, G \models \{ C', \sigma' \} \preceq (C', \Sigma') \circ (u', w_{s'}, w', H + 1) \downarrow \xi' \ p$, and
   (D) $(\{ (\sigma, \Sigma), (\sigma', \Sigma') \}, k) \models G_t \ast \text{True}, \text{and}$
   (E) either $u' < k, u'$
      or $u' = u$ and $k = 0$ and $w_{s'} <_{R+1} w_{s1}$ and $w_1 = w_1$,
      or $u' = u$ and $k = 0$ and $w_{s'} = w_{s1}$ and $w_1 = w_1$ and $\xi_1 \neq \emptyset$ and $\xi_1 \subseteq \xi'$; and
   (F) if $(\{ (\sigma, \Sigma), (\sigma', \Sigma') \}) \models \{ (D_1) \} \ast \text{True},$ then $w_{s'}' <_{R+1} w_{s1}$.

Since $(\{ (\sigma, \Sigma), (\sigma', \Sigma') \}) \models G_t \ast \text{True}, (\sigma, \Sigma) \models J \ast \text{true}, J \ast I, I \ast G$ and $\text{Sta}(J, G \lor R)$, we know $(\sigma', \Sigma') \models J \ast \text{true}$.

Suppose $k'_1 = f(\sigma', \Sigma')$. Since $J \Rightarrow (R, G; D' \downarrow \to Q)$, we can prove

$$k = 0 \Rightarrow k' \leq k_1 .$$

Also we have

$$(\sigma', \Sigma') \models Q_t \ast \text{true} \lor (\exists t' \neq t. \text{Enabled}(D'_t) \ast \text{true}) .$$

Let

$$\xi_0 = \{ t' | (t' \neq t) \land (\sigma', \Sigma') \models \text{Enabled}(D'_t) \ast \text{true} \} .$$

Since for any $t' \ast t$ we have $G_t \Rightarrow R_{t'}$, and since $\text{wffAct}(R, D') \circ D' \leq D$ and $\text{Enabled}(D) \Rightarrow I$, we can prove:

$$k = 0 \Rightarrow \xi_0 \leq \xi_0 .$$

Let

$$w_{s'} = (0, 0) :: \text{inhead}(w_{s1}', (w_1, 1)) .$$
We know if $ws' \prec \mathcal{H} ws_1$, then $ws' \prec \mathcal{H}+1 ws$. If $w'_1 = w_1$, let $w' = w$; otherwise let $w' = w'_1$. Thus we know $w'_1 \leq w'$.
Suppose $\text{head}(ws'_1) = ((n'_1, n'_2), n_3)$.

- If $\xi_0 \neq \emptyset$, let
  
  \[ ws' = ((0, 0), 0) : \text{inhead}(ws'_1, ((k'_1, w_1), -n'_3)) \]  
  and $\xi' = \xi_0 \cup \xi_1$.

  Then by the co-induction hypothesis, we know
  \[ D, R, G \models I \quad \text{(C1)} \quad \text{while} \quad (B)(C), \sigma') \leq (C', \Sigma') \circ (u', ws', ws', w', H + 1) (w'_1 \psi' p \land \neg B). \]

  Also, if $ws'_1 \leq \mathcal{H} ws_1$, then $ws' \leq \mathcal{H}+1 ws$. Thus, we know: either $u' < u$, or $u' = u$ and $k = 0$ and $ws' \leq \mathcal{H}+1 ws$ and $w' = w$.

  For the case that $u' = u$ and $k = 0$ and $ws' = ws$ and $w' = w$, we know
  
  \[ ws = ((0, 0), 0) : \text{inhead}(ws_1, ((k, w_1), -n_3)) \]  
  and $\xi_0 \neq \emptyset$ and $\xi = \xi_0 \cup \xi_1$,  

  thus $\xi \neq \emptyset$.

- Suppose $ws'_1 < \mathcal{H} ws_1$, since $ws' = ws$, we know $n'_3 < n_3$. Then, from the definition of $< \mathcal{H}$, we know $ws_1 = ((n_1, n_2), n_3)$, thus $\xi_1 = \emptyset$. Thus $\xi \subseteq \xi'_1$.

- Suppose $ws'_1 = ws_1$. Then we know $\xi_1 \subseteq \xi'_1$.

Thus $\xi \subseteq \xi'_1$.

- If $\xi_0 = \emptyset$, then we know $(\sigma', \Sigma') = Q_t \ast$ true. Since $\xi_0 \subseteq \xi'_0$, we know $\xi_0 = \emptyset$. Thus we know $(\sigma, \Sigma) \models I \ast$ true and $ws = ((0, 0), 0) : \text{inhead}(ws_1, ((k, w_1), 1))$. Also we have $\xi = \xi_0 \cup \xi_1 = \xi_1$. Let

  \[ ws' = ((0, 0), 0) : \text{inhead}(ws_1, ((k'_1, w_1), 1)) \]  
  and $\xi' = \xi_0 \cup \xi_1 = \xi'_1$.

By the co-induction hypothesis, we know

\[ D, R, G \models I \quad \text{(C1)} \quad \text{while} \quad (B)(C), \sigma') \leq (C', \Sigma') \circ (u', ws', ws', w', H + 1) (w'_1 \psi' p \land \neg B). \]

Also, if $ws'_1 \leq \mathcal{H} ws_1$, then $ws' \leq \mathcal{H}+1 ws$. Thus, we know: either $u' < u$, or $u' = u$ and $k = 0$ and $ws' \leq \mathcal{H}+1 ws$ and $w' = w$.

For the case that $u' = u$ and $k = 0$ and $ws' = ws$ and $w' = w$, we know $ws'_1 = ws_1$, thus $\xi \neq \emptyset$. Since $\xi \subseteq \xi'_1$, we know $\xi \subseteq \xi'_1$.

(II) If $C_1 = \text{skip}$ and $(C_1) \quad \text{while} \quad (B)(C), \sigma \ast \sigma_F \rightarrow t \ast \text{while} \quad (B)(C), \sigma \ast \sigma_F$, from $D, R, G \models I \quad (C_1, \sigma), (u, w_{s1}, w_1, H) \psi\xi, p$, we know there exist $C'$ and $\Sigma'$ such that

(A) $(C, \Sigma, \psi_F) \rightarrow t \ast (C', \Sigma', \psi_F)$, \n
(B) $((\sigma), (\xi), (u, w_{s1}, w_1, H)) \models p$, and

(C) $ws_{s1} = ((0, 0), 0)$ and $ws_1 = ((0, 0), 0)$ and $\xi_1 = \emptyset$, and

(D) $((\sigma), (\xi), (\Sigma')) = G_t \ast$ true.

Since $(\sigma, \Sigma) \models J \ast$ true, $J \Rightarrow I$, $I \Rightarrow G$ and $\text{Stab}(J, G \ast R)$, we know $(\sigma, \Sigma') \models J \ast$ true.

Suppose $k'_t = f(\sigma, \Sigma')$. Since $J \Rightarrow (R, G) \ast D' \ast t(Q)$, we can prove

$k'_t \leq k_t$.

Also we have

$(\sigma, \Sigma') = Q_t \ast$ true $\lor (\exists t' \neq t. \text{Enabled}(D'_t) \ast$ true).

Let

$\xi'_0 = \{ t' \mid t' \neq t \land (\sigma, \Sigma') \models \text{Enabled}(D'_t) \ast$ true $\}$.

Since for any $t'$ such that $t' \neq t$ we have $G_t \Rightarrow R_t$, and since $w_{s\text{ff}}(R, D') \ast D' \leq D$ and $\text{Enabled}(D) \Rightarrow I$, we can prove:

$\xi_0 \leq \xi'_0$.

Let

$\downarrow ws' = ((0, 0), 0) : \text{inhead}(ws_1, (w_1, 0))$.

Thus $ws' \leq \mathcal{H}+1 ws$. Let

$\downarrow ws' = ((0, 0), 0) : \text{inhead}(ws_{s1}, ((k'_1, w_1), 0))$.

Thus $ws' \leq \mathcal{H}+1 ws$. Also, if $ws' = ws$, then $ws = ((0, 0), 0) : \text{inhead}(ws_{s1}, ((k, w_1), -n_3))$ and $\xi_0 \neq \emptyset$ and $\xi = \xi_0 \cup \xi_1$, thus $\xi \neq \emptyset$ and $\xi \subseteq \xi'_0$.

Below we prove:

\[ D, R, G \models I \quad \text{while} \quad (B)(C), \sigma \leq (C', \Sigma') \circ (u, ws', ws', w', H, 1) (w'_1 \psi' p \land \neg B). \]

(B.2)

Proof: By co-induction. Suppose $\sigma = (s, h)$. Since $p \Rightarrow (B = B) \ast I$, we know

$(\sigma, \Sigma') \models I \ast$ true, and either $[B]_s = \text{true}$ or $[B]_s = \text{false}$.

We only need to prove the following (I)(2)(3)(4)(5).

(I) For any $t' \in \xi'_0$, we have $(\sigma, \Sigma') \models \text{Enabled}(D'_t) \ast$ true.

Proof: Immediate from $D' \leq D$.

(2) If while $(B)(C) = \text{skip}$, then...

Proof: It is vacantly true.

(3) For any $\sigma_F$, while $(B)(C), \sigma \ast \sigma_F$ $\rightarrow t \ast$ $\text{abort}$.

Proof: It holds because $[B]_s$ is not undefined.

(4) If while $(B)(C), \sigma \ast \sigma_F \rightarrow t \ast (C', \sigma')$, then there exist $\sigma', \Sigma', \Sigma''$, $k, u''', ws''', ws'''$ and $\xi' \ast$ such that

(a) $\sigma' = \sigma \ast \sigma_F$, and

(b) $(\Sigma', \Sigma'' \ast \sigma_F) \rightarrow t \ast (\Sigma', \Sigma'' \ast \sigma_F)$, and

(c) $D, R, G \models I \ast (C', \sigma') \leq (C'', \Sigma'') \circ (u'', ws'', ws'', w'', H, 1) (w'_1 \psi' p \land \neg B)$, and

(d) $((\sigma, \Sigma'), (\sigma'), (\Sigma')) \models G_t \ast$ True, and
(e) either \( u'' <_k u \),
or \( u'' = u \) and \( k = 0 \) and \( w'' = w'' \) and \( w'' = w \),
or \( u'' = u \) and \( k = 0 \) and \( w'' = w'' \) and \( w'' = w \) and \( \xi_0 \neq 0 \) and \( \xi_0 \subseteq \xi' \); and

(f) if \(((\sigma', \Sigma'), (\sigma', \Sigma'')) \models \{D_i\} \ast True\), then \( w'' <_{n+1} w' \).

Proof: We have two cases depending on whether \([B]_s = \text{true}\) or \(false\).

- If \([B]_s = true\), we know (while \((B)\{C\}, \sigma \cup \sigma_F) \rightarrow \rightarrow (C; while \((B)\{C\}, \sigma \cup \sigma_F\). Also we know \(((\sigma, \Sigma'), (u, w_1), C') = p_k \land B\).

- If \(\xi_0 \neq 0\), we have two cases below:

  - If \(\exists (\sigma, \Sigma') = Enabled(D_i) \ast true\), we know
    \[
    \{\sigma, \Sigma'\}, (u, w_1), C') = p_k \land B \land Enabled(D_i) \ast true
    \]
    Since \(p \land B \land Enabled(D_i) \ast true \Rightarrow p' \ast (w'(1) \land emp)\), we know there exists \(w_1\) such that \(w_1 < w_1\) and
    \[
    \{\sigma, \Sigma'\}, (u, w_1), C') = p_k.
    \]
    From \(D, R, G \models \{p'\}C(p)\) and \(height(C) = H\), let \(w_1 = (0, C)\) and \(w_1 = (0, C)\), we get:
    \[
    D, R, G \models (C, \sigma) \leq (C', \Sigma') \circ (w_1, w_1, C), H) \cup_0 p.
    \]
    Let \(w'' = ((0, 0), 0) :: ((k', w_1), 0)\) and \(w'' = ((0, 0), 0) :: ((k', w_1), C) + 1\). By the co-induction hypothesis, we know
    \[
    D, R, G \models (C, while (B)\{C\}, \sigma) \leq (C', \Sigma') \circ (w_1, w_1, C), H) \cup_0 p.
    \]
    Also we have \(w'' <_{n+1} w'\) and \(w'' <_{n+1} w'\). Since \(\bar{i} \ast true\), we can prove:
    \[
    \{\sigma, \Sigma'\}, (\sigma, \Sigma'), 0) = G_t \ast True
    \]

- Otherwise, from \(p \land B \Rightarrow p'\), we know
    \[
    \{\sigma, \Sigma'\}, (u, w_1), C') = p_k,
    \]
    From \(D, R, G \models \{p'\}C(p)\) and \(height(C) = H\), let \(w_1 = (0, C)\) and \(w_1 = (0, C)\), we get:
    \[
    D, R, G \models (C, \sigma) \leq (C', \Sigma') \circ (u, w_1, w_1, C), H) \cup_0 p.
    \]
    Let \(w'' = ((0, 0), 0) :: ((k', w_1), 0)\) and \(w'' = ((0, 0), 0) :: ((k', w_1), C) + 1\). By the co-induction hypothesis, we know
    \[
    D, R, G \models (C, while (B)\{C\}, \sigma) \leq (C', \Sigma') \circ (w_1, w_1, C), H) \cup_0 p.
    \]
    Also we have \(w'' = w''\) and \(\xi_0 = 0\). Since \((\sigma, \Sigma') \models I \ast true\), we can prove:
    \[
    \{\sigma, \Sigma'\}, (\sigma, \Sigma'), 0) = G_t \ast True
    \]

- If \(\xi_0 = 0\), then we know \((\sigma, \Sigma') \models Q_t \ast true\). Thus we know
    \[
    \{\sigma, \Sigma'\}, (u, w_1), C') = p_k \land B \land Q_t \ast true
    \]
    Since \(p \land B \land Q_t \ast true \Rightarrow p' \ast (\cup(1) \land emp)\), we know there exists \(w_1\) such that \(w_1 < w_1\) and
    \[
    \{\sigma, \Sigma'\}, (u, w_1), C') = p_k.
    \]
    From \(D, R, G \models \{p'\}C(p)\) and \(height(C) = H\), let \(w_1 = (0, C)\) and \(w_1 = (0, C)\), we get:
    \[
    D, R, G \models (C, \sigma) \leq (C', \Sigma') \circ (u, w_1, w_1, C), H) \cup_0 p.
    \]
    Let \(w'' = ((0, 0), 0) :: ((k', w_1), C) + 1\). By the co-induction hypothesis, we know
    \[
    D, R, G \models (C, while (B)\{C\}, \sigma) \leq (C', \Sigma') \circ (u, w_1, w_1, C), H) \cup_0 p.
    \]
    Also we have \(w'' = w''\). Since \((\sigma, \Sigma') \models I \ast true\), we can prove:
    \[
    \{\sigma, \Sigma'\}, (\sigma, \Sigma'), 0) = G_t \ast True
    \]

- If \([B]_s = false\), we know (while \((B)\{C\}, \sigma \cup \sigma_F) \rightarrow \rightarrow (\text{skip}, \sigma \cup \sigma_F)\). By (\text{skip}) rule, let \(w'' = ((0, 0), 0)\) and \(w'' = ((0, 0), 0)\), we know
    \[
    D, R, G \models (\text{skip}, \sigma) \leq (C', \Sigma') \circ (u, w_1, w_1, C), H) \cup_0 p \land B.
    \]
    Also we have \(w'' <_{n+1} w''\). Since \((\sigma, \Sigma') \models I \ast true\), we can prove:
    \[
    \{\sigma, \Sigma'\}, (\sigma, \Sigma'), 0) = G_t \ast True
    \]

(5) If \(((\sigma, \Sigma'), (\sigma', \Sigma'')) \models R_t \ast Id\), then there exist \(u, w_1, w_1, \xi_0, \xi_0\), \(\xi_0\), and \(\xi_0\) such that

- (a) \(D, R, G \models (\text{while} (B)\{C\}, \sigma) \leq (C', \Sigma'') \circ (u, w_1, w_1, C), H) \cup_0 p \land B\), and
- (b) \(\xi_0 = \{u' | u' \in \xi_0 \land (\{\sigma, \Sigma'\}, (\sigma', \Sigma'')) \models \{D_i\} \ast Id\} \land (k = 0 \Rightarrow \xi_0 \subseteq \xi_0 \subseteq \xi_0)\), and
- (c) if \(k = 0\), then either \(w'' <_{n+1} w''\) and \(w'' = w\), or \(w'' = w''\) and \(w'' = w\) and \(\xi_0 = \emptyset\); and
- (d) if \(k = 0\) and \((\sigma, \Sigma') \models Enabled(D_i) \ast true\), then \(w'' <_{n+1} w''\).

Proof: Since \(\text{Sta}(p, R \ast Id)\), we know there exist \(u'\) and \(w_1\) such that
\[
\{\sigma', \Sigma''\}, (u', w_1), C') = p_k \land u' \approx_k u \land k = 0 \Rightarrow w_1 = w_1.
\]
Also we know \(\{\sigma', \Sigma''\} = J \ast true\).

Suppose \(k'' = f(\sigma', \Sigma'')\). Since \(J \Rightarrow (R, G : D'_{\Delta_S} Q)\), we can prove \(k'' \leq k''\).
Let
\[\xi_0' = \{t' | (t' \neq t) \land ((\sigma', \Sigma') \models \text{Enabled}(D_{t'}) \land \text{true})\}\] and \[\xi_d = \{t' | (t' \in \xi_1) \land (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle D_t \rangle \land \text{false})\}\].

Since enabled(D) \Rightarrow I, D' \subseteq D and wffAct(R, D'), we can prove:
\[k = 0 \implies \xi_0' \subseteq \xi_0\] and \[\xi_d \subseteq \xi_0\].

If \(w_1 = w_1\), let \(w'' = w\); otherwise let \(w'' = w_1\). Thus we know \(w'' \leq w_1\). Let
\[w'' = ((0,0), 0)\]::inhead(ws1, ((k''_1, w_1'), 0))

By the co-induction hypothesis, we know
\[D, R, G \models (\text{while } (B) \{C\}, \sigma') \subseteq (C', \Sigma'') \circ (u, ws''', ws', w'', H + 1) \psi_{\xi''} p \land \neg B\]

Also we know: if \(k = 0\), then \(ws'' \leq H + 1\) and \(w'' = w\).

If \(k = 0\) and \(\xi_d \neq \emptyset\), then there exists \(t'\) such that \(t' \in \xi_1\) and \(((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle D_t \rangle \land \text{Id}\). Since \(D' \subseteq D\), we can prove
\[((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle D_t \rangle \land \text{Id}\]

Since \(J \Rightarrow (R, G; D' \downarrow Q)\), we know for any \(t' \neq t\), \(\sigma'\) and \(\Sigma'\), if \(((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle D_t \rangle \land \text{Id}\), then \(f(\sigma', \Sigma') < k_1\). Thus we can prove:
\[k''_1 < k_1\].

Thus \(ws'' < H + 1\) and \(ws'\) holds.

\(\text{Thus we have proved } [B.2]\).
• If $\xi_d \neq \emptyset$, let
  $\omega_s' = ((0, 0), 0) :: \text{inhead}(\omega s_1', ((k_1', w_1), -n_3'))$ and $\xi' = \xi_d' \cup \xi_1'$.

Then by the co-induction hypothesis, we know

$D, R \vdash (C_1; \text{while } (B) \{C\}, \sigma') \preceq (C, \Sigma') \circ (u, \omega s', \omega s', \omega', H + 1) \models_{\xi'} p \land \neg B$.

If $k = 0$, we know $\xi \xi_d \subseteq \xi', \omega s' \leq H + 1 \omega s$ and $\omega' = w$.

For the case $k = 0$ and $\xi_0 \neq \emptyset$, we know $\xi_0 \neq \emptyset$ or $\xi_d' \neq \emptyset$. If $\xi_d' \neq \emptyset$, we know $\omega s' \leq H + 1 \omega s$. If $\xi_d' \neq \emptyset$, we know $\xi_1 \neq \emptyset$ and $\omega s_1' \leq H + 1 \omega s_1$.

Thus $|\omega s_1| > 1$. From the definition of $<_{H}$, we can prove: $\omega s' \leq H + 1 \omega s$.

• If $\xi_d' = \emptyset$, then we know $(\sigma', \Sigma', i) \models Q_1 *$ true. Let
  $\omega s' = ((0, 0), 0) :: \text{inhead}(\omega s_1', ((k_1', w_1), 1))$ and $\xi' = \xi_0 \cup \xi_1' = \xi_1'$.

By the co-induction hypothesis, we know

$D, R \vdash (C_1; \text{while } (B) \{C\}, \sigma') \preceq (C, \Sigma') \circ (u, \omega s', \omega s', \omega', H + 1) \models_{\xi'} p \land \neg B$.

If $k = 0$, we know $\xi \xi_d \subseteq \xi', \omega' = w$ and $\omega s_1' \leq H + 1 \omega s_1$.

• If $\xi_0 \neq \emptyset$ and $\omega s = ((0, 0), 0) :: \text{inhead}(\omega s_1, ((k_1, w_1), -n_3))$, since $\xi_0 \xi_0' \subseteq \xi_0'$, we can prove $\xi_0' \neq \emptyset$.

Then we know $k' < k_1$. Thus, if $k = 0$, then $\omega s' \leq H + 1 \omega s$.

• If $(\sigma', \Sigma', i) \models Q_1 *$ true and $\omega s = ((0, 0), 0) :: \text{inhead}(\omega s_1, ((k_1, w_1), 1))$, we know: if $k = 0$, $\omega s' \leq H + 1 \omega s$. If $\xi_d' \neq \emptyset$, then $\xi_0' \neq \emptyset$ or $\xi_0' \neq \emptyset$, thus $\omega s' \leq H + 1 \omega s$.

Thus we are done.

\[ \square \]

### B.2.2 The Atom Rule

**Lemma 15 (ATOM-Sound).** If

1. $\models_{\text{atom}} \{p\} \{q\}$;
2. $q' \Rightarrow k q$;
3. $\{(\{p\} \times_k \{q\}\} \Rightarrow G \ast True$;
4. $p \Rightarrow q \Rightarrow I \ast true$;
5. Locality(C);

then $D, I, G \models \{p\} \{q\}$. $\models_{\text{atom}} (C, \sigma)$.

**Proof.** Let $H = \text{height}(\{C\}) = 1$. We know $|\{C\}| = 1$. Let

$\omega s = ((0, 0), 1)$ and $\omega s = (0, 1)$ and $\xi = \emptyset$.

Below we prove: for any $t$, for any $\sigma, \Sigma, u, w$ and $\Sigma$, if $(\sigma, \Sigma, (u, w)) \models p, then$

$D, I, G \models \{C\}, \sigma \preceq (\Sigma, \Sigma') \circ (u, \omega s, \omega s, \omega, H) \models_{\xi} q$.

By co-induction. Since $p \Rightarrow I \ast true$, we know $(\sigma, \Sigma) \models I \ast true$. From the premises, we can prove:

$(C, \sigma) \not\Rightarrow_{\ast} \text{abort} \land (C, \sigma) \not\Rightarrow_{\ast} \text{\textemdash} \cdot$.

We only need to prove the following (1)(2)(3)(4)(5)(6).

(1) For any $t', t \in \xi$, we have $(\sigma, \Sigma) \models \text{Enabled}(\{D_t\}) \ast true$.

**Proof:** It is vacuously true.

(2) If $\{C\} = \text{skip}$, then ...

**Proof:** It is vacuously true.

(3) If $\{C\} = \text{E[\text{return} E]}$, then ...

**Proof:** It is vacuously true.

(4) For any $\sigma, \Sigma, \{C\}, \sigma \vdash \sigma F$ and $\Sigma_F$, if $(\{C\}, \sigma \vdash \sigma F) \Rightarrow_{\ast} (C', \sigma'''),$ then there exist $\sigma', \Sigma', k, u', \omega s', \omega s', \omega'$ and $\xi'$ such that

(a) $\sigma'' \Rightarrow \sigma' \vdash \sigma F$,

(b) $(\sigma', \Sigma' \subseteq \Sigma_F)$, and

(c) $\{C', \Sigma' \vdash (\sigma', \Sigma') \circ (u', \omega s', \omega s', \omega', H) \models_{\xi'} q$, and

(d) $(\sigma', \Sigma', \sigma', \Sigma')$, $k \models G_1 \ast True$.

(e) either $u' <_k u$, or $u' = k = 0$ and $\omega s' \leq H \omega s$ and $\omega' = w$,

or $u' = u$ and $k = 0$ and $\omega s' = \omega s$ and $\omega' = w$ and $\xi' \neq \emptyset$ and $\xi' \subseteq \xi'$.

(f) if $(\sigma, \Sigma), (\sigma', \Sigma')$, $k \models \{D_t\} \ast True$, then $\omega s' \leq H \omega s$.

**Proof:** By the operational semantics, we know $C'$ must be $\text{skip}$ and

$(C, \sigma \vdash \sigma F) \Rightarrow_{\ast} (\text{skip}, \sigma''')$.

By Locality(C), we know there exists $\sigma'$ such that $\sigma''' = \sigma' \vdash \sigma F$ and
\((C, \sigma) \rightarrow_\ast^\dagger \text{(skip, } \sigma')\).

From \(\models_\ast [p]C[q']\), we know
\(((\sigma', \Sigma), (u, w), C) \models q'\).

From \(q' \Rightarrow_k q\), we know: there exist \(u', \Sigma', C'\) and \(\Sigma'\) such that
(A) \((C, \Sigma \uplus \Sigma_F) \rightarrow_\ast (C', \Sigma' \uplus \Sigma_F)\), and
(B) \(((\sigma', \Sigma'), (u', \Sigma), C') \models q_\ast\), and
(C) either \(u' <_k u\), or \(k = 0\) and \(u' = u\) and \(w' = w\).

Thus we know
\(((\sigma, \Sigma), (\sigma', \Sigma'), k) \models ([p] \& [q])\).

Since \((\llbracket p \rrbracket \& [q]) \Rightarrow G \ast \text{True}\), we know
\(((\sigma, \Sigma), (\sigma', \Sigma'), k) \models G_\ast \Rightarrow \text{True}\).

By (skip) rule, let
\(\omega s' = ((0, 0), 0)\) and \(\omega s' = (0, 0)\).

we know
\(\mathcal{D}, [I], G \models_\ast \text{(skip, } \sigma') \preceq (C', \Sigma') \circ (\omega, \omega s', \omega s', \omega')\),

Also we know \(\omega s' \prec_M \omega s\) and \(\omega s' \prec_M \omega s\).

(6) If \(((\sigma, \Sigma), (\sigma', \Sigma'), k') \models [I] \ast \text{Id}\), then there exist \(u', \omega s', \omega s', \Sigma'\) and \(\xi_d\) such that
(a) \(\mathcal{D}, R, G \models_\ast ((C'), \sigma') \preceq (C', \Sigma') \circ (\omega, \omega s', \omega s', \omega')\),
(b) \(\xi_d = \{\xi' \mid (\xi' \in \xi) \land (((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle \mathcal{D}_\ast \rangle \ast \text{Id})\}\) and \(k' = 0 \Rightarrow \xi' \preceq (\Sigma' \subseteq \xi')\) and \(u' \preceq_M u\), and
(c) if \(k' = 0\), then either \(\omega s' \preceq_M \omega s\) and \(w' = w\), or \(\omega s = \omega s\) and \(w' = w\) and \(\xi_d = \emptyset\); and
(d) if \(k' = 0\) and \((\sigma, \Sigma) \models \text{Enabled}(\mathcal{D}_\ast) \ast \text{true}\), then \(\omega s' \preceq_M \omega s\).

Proof: We know \(\sigma' = \sigma\) and \(\Sigma' = \Sigma\) and \(k' = 0\).

By the co-induction hypothesis, we get
\(\mathcal{D}, [I], G \models_\ast ((C), \sigma') \preceq (C, \Sigma) \circ (u, \omega s, \omega s, w, H)\),

Thus we are done.
B.3 Common Simulation and Instantiations

We introduce a state-updating function $\Delta$ to describe the high-level state updates which may delay other threads. That is, at any state $\Delta_1(S)$, another thread $t'$ is possible to make stuttering steps.

$$\Delta \in \text{ThrdID} \times \text{Mem} \to \text{Mem}$$

**Definition 16** (Common Simulation for Object). \( \Pi \preceq \Pi' \) iff there exists $\Delta_1, D, R, G$ such that $D, R, G \models \Delta_1 \{ P \} \Pi \preceq \Pi'$ holds.

\[ D, R, G \models \Delta_1 \{ P \} \Pi \preceq \Pi' \text{ iif, for any } f \in \text{dom}(\Pi) \text{, for any } \sigma \text{ and } \Sigma \text{, for any } t, \text{ if } \Pi(f) = (x, C), \Pi'(f) = (y, C) \text{ and } (\sigma, \Sigma) \models P \ast \text{own}(x) \ast \text{own}(y)/(x = y), \text{ there exist two well-founded metrics } M \text{ and } N \text{ and a set } \xi \in \mathcal{P}(\text{ThrdID}) \text{ such that} \]

\[ D, R, G \models \Delta_1 (C, \sigma) \preceq (C, \Sigma) \circ (M, M) \downarrow_\xi (P \ast \text{own}(x) \ast \text{own}(y)). \]

Here $D, R, G \models \Delta_1 (C, \sigma) \preceq (C, \Sigma) \circ (M, M) \downarrow_\xi Q$ is co-inductively defined as follows.

Whenever $D, R, G \models \Delta_1 (C, \sigma) \preceq (C, \Sigma) \circ (M, M) \downarrow_\xi Q$ holds, then the following hold:

1. Suppose $\sigma = (s, h)$. Then $\xi \subseteq s(\text{TIDS})$ and $t \notin \xi$.
2. For any $t' \in \xi$, we have $(\sigma, \Sigma) \models \text{Enabled}(D_{t'}) \ast \text{true}$.

$\Pi \models s(\text{TIDS})$ and $t \notin \xi$.

(2) If $C = E[\text{return } E]$, then for any $\Sigma_F$ such that $\Sigma \models \Sigma_F$, there exist $E$ and $\Sigma'$ such that

\[ (a) \ (\Sigma, \Sigma \uplus \Sigma_F) \rightarrow^* (\text{return } E, \Sigma' \uplus \Sigma_F) \text{, and} \]

\[ (b) \ (\sigma, \Sigma') \models Q_1 \text{ and } [E]_{\sigma, s} = [E]_{\Sigma', s} \text{, and} \]

\[ (c) \ (\sigma, \Sigma), (\sigma', \Sigma') \models G_1 \ast \text{true}. \]

(3) For any $\Sigma_F \circ (C, \sigma \uplus \Sigma_F) \rightarrow^* \ast$ abort.

(4) For any $C', \sigma', \Sigma_F$, if $C, \sigma \uplus \Sigma_F \rightarrow^{*} (C', \sigma'' \uplus \Sigma') \circ (\Sigma \uplus \Sigma_F)$, then there exist $\sigma', n, C', \Sigma', k, M', \text{ and } \xi'$ such that

\[ (a) \ (\sigma') = (\sigma' \uplus \Sigma_F), \text{ and} \]

\[ (b) \ (\Sigma, \Sigma \uplus \Sigma_F) \rightarrow^* (C', \Sigma' \uplus \Sigma_F); \text{ and} \]

if $k > 0$, then there exist $n_1, n_2$ and $C'_{\sigma'}$ such that $n = n_1 + n_2 > 0$ and $(\Sigma, \Sigma \uplus \Sigma_F) \rightarrow^* (C', \Sigma' \uplus \Sigma_F) \rightarrow^* (\Sigma'', \Sigma' \uplus \Sigma_F)$ and

\[ (C'', \Sigma, \Sigma \uplus \Sigma_F) \rightarrow^* (\Sigma', \Sigma' \uplus \Sigma_F) \downarrow_\xi Q \text{, and} \]

\[ (D, R, G) \models \Delta_1 (C', \sigma') \preceq (C', \Sigma') \circ (M', M) \downarrow_\xi Q \text{, and} \]

\[ (c) \ (\sigma', \Sigma') \models G_1 \ast \text{true}. \]

(5) For any $k < 0$, or $M' < M$, $\xi = \emptyset$ and $\xi' < \xi$ such that

\[ \xi_{\text{dom}} \rightarrow_{\ast} \emptyset \text{, or } \xi' = \mathbb{V} \text{ and } \xi \subseteq \xi' \text{, and} \]

\[ \text{either } M' < M, \text{ or } \xi' \neq \emptyset \text{ and } (\sigma, \Sigma) \models \text{Enabled}(D_{\xi}) \ast \text{true}, \text{ then } M' \leq M. \]

B.3.1 Instantiating the Common Simulation

**Lemma 17** (Q in Fig. [13]). Suppose $R \models [R]_0$ and $G \models [G]_0$. If $D, R, G \models \{ P \} \Pi : \Gamma$, then $D, R, G \models \emptyset \{ P \} \Pi \models \Gamma$.

**Proof.** For any $f \in \text{dom}(\Pi)$, for any $\sigma$ and $\Sigma$, for any $t$, if $\Pi(f) = (x, C)$, $\Gamma(f) = (y, C)$ and $(\sigma, \Sigma) \models P_t \ast \text{own}(x) \ast \text{own}(y)/(x = y)$, from $D, R, G \models \{ P \} : \Gamma$, we know there exist three well-founded metrics $u, M$ and $N$ and a set $\xi \in \mathcal{P}(\text{ThrdID})$ such that

\[ D, R, G \models \{ (C, \sigma) \} \leq (u, M, M) \downarrow_\xi (P \ast \text{own}(x) \ast \text{own}(y)). \]

We want to prove: there exist two well-founded metrics $M$ and $N$ and a set $\xi \in \mathcal{P}(\text{ThrdID})$ such that

\[ D, R, G \models \emptyset \{ (C, \sigma) \} \leq (u, M, M) \downarrow_\xi (P \ast \text{own}(x) \ast \text{own}(y)). \]

We only need to prove the following:

\[ D, R, G \models (C, \sigma) \leq (u, M, M) \downarrow_\xi (P \ast \text{own}(x) \ast \text{own}(y)). \]

By co-induction.

1. Suppose $\sigma = (s, h)$. Then $\xi \subseteq s(\text{TIDS})$ and $t \notin \xi$. For any $t' \in \xi$, we have $(\sigma, \Sigma) \models \text{Enabled}(D_{t'}) \ast \text{true}$.

**Proof.** Immediate.

(2) If $C = E[\text{return } E]$, then for any $\Sigma_F$ such that $\Sigma \models \Sigma_F$, there exist $E$ and $\Sigma'$ such that

\[ (a) \ (\Sigma, \Sigma \uplus \Sigma_F) \rightarrow^* (\text{return } E, \Sigma' \uplus \Sigma_F) \text{, and} \]

\[ (b) \ (\sigma, \Sigma') \models Q_1 \text{ and } [E]_{\sigma, s} = [E]_{\Sigma', s} \text{, and} \]

\[ (c) \ (\sigma, \Sigma), (\sigma', \Sigma') \models G_1 \ast \text{true}. \]

**Proof.** Immediate.

(3) For any $\sigma_F \circ (C, \sigma \uplus \Sigma_F) \rightarrow^* \ast$ abort.

**Proof.** Immediate.

(4) For any $C', \sigma'', \Sigma_F$, if $\{ (C, \sigma \uplus \Sigma_F) \rightarrow^{*} (C', \sigma'') \} \circ (\Sigma \uplus \Sigma_F)$, then there exist $\sigma', n, C', \Sigma', k, M'$ and $\xi'$ such that
Thus we are done.

From (C), by the dictionary order, we know

\[
\text{either } (u', M') < (u, M), \text{ or } (u', M') = (u, M) \text{ and } \xi \neq \emptyset \text{ and } \xi \subseteq \xi'.
\]

From (D), since \( G \Rightarrow [G]_0 \), we know

\[
((\sigma, \Sigma), (\sigma', \Sigma'), 0) \models G_* \text{ true.}
\]

(5) For any \( k, \sigma' \) and \( \Sigma' \), if \(( (\sigma, \Sigma), (\sigma', \Sigma'), k ) \models R_* \text{ ld}, \) then there exist \( M', M', \xi_d \) and \( \xi' \) such that

(a) \( D, R, G \models (C, \sigma') \preceq (C, \Sigma') \circ (u', M') \psi_{\xi'} Q, \)

(b) \( \xi_d = \{ t' \mid ( t' \in \xi ) \land ((\sigma, \Sigma), (\sigma', \Sigma')) \models (D_{t'}) \text{ ld} \} \) and \( k = 0 \Rightarrow \xi \subseteq \xi' \)

(c) if \( k > 0, \) then for any \( t' \neq t \) and \( \Sigma \uplus \Sigma \) we have \(( C, \Delta_{\Sigma'} \circ \Sigma_F ) \models (\Delta_{\Sigma} \circ \Sigma_F ) ; \) otherwise, \( t' < (u, M), \) or \( M' = (u, M) \) and \( \xi_d = \emptyset; \) and

(d) if \( k = 0 \) and \(( \sigma, \Sigma ) \models \text{ Enabled}(D_{t'}) \text{ true}, \) then \( M' \leq M. \)

Proof: Since \( R \Rightarrow [R]_0, \) we know

\[
((\sigma, \Sigma), (\sigma', \Sigma'), 0) \models R_* \text{ ld } .
\]

Then, since \( D, R, G \models (C, \sigma) \preceq (C, \Sigma) \circ (u, M, M) \psi_{\xi} Q, \) we know there exist \( u', M', M', \xi_d \) and \( \xi' \) such that

(a) \( D, R, G \models (C, \sigma') \preceq (C, \Sigma') \circ (u', M', M') \psi_{\xi'} Q, \)

(b) \( \xi_d = \{ t' \mid ( t' \in \xi ) \land ((\sigma, \Sigma), (\sigma', \Sigma')) \models (D_{t'}) \text{ ld} \} \) and \( \xi \subseteq \xi' \)

(c) if \( M' < M, \) or \( M' = M \) and \( \xi_d = \emptyset; \) and

(d) if \( \sigma, \Sigma \models \text{ Enabled}(D_{t'}) \text{ true}, \) then \( M' \leq M. \)

From (A), by the co-induction hypothesis, we know

\[
D, R, G \models (C, \sigma') \preceq (C, \Sigma') \circ (u', M', M') \psi_{\xi'} Q.
\]

From (C), by the dictionary order, we know

\[
\text{either } (u', M') < (u, M), \text{ or } (u', M') = (u, M) \text{ and } \xi_d = \emptyset.
\]

Thus we are done.

**Lemma 18** (§ in Fig. 18). Suppose \( 1 \notin \text{ fr}(D, R, G, P, \Pi, \Gamma). \)

If \( D, R, G \models \{ P \} \Pi : \Gamma, \) then \( D, R * [1 = 0], G * [1 = 0] \models \Delta: \{ P * [1 = 0] \} \Pi \models \psi_{\text{wr}(\Gamma)}. \) Here

\[
\Delta \Sigma(1, \Gamma) \triangleq \begin{cases} 
\Sigma(1, \Gamma) & \text{if } \Sigma(1) = 0 \\
\text{undefined} & \text{if } 1 \notin \text{ dom}(\Sigma) \text{ or } \Sigma(1) \neq 0
\end{cases}
\]

Proof: For any \( f \in \text{ dom}(\Pi), \) for any \( \sigma \) and \( \Sigma, \) for any \( \text{ t, } \) if \( \Pi(f) = (x, C), \Gamma(f) = (y, (C); \text{ return } E) \) and \( \sigma, \Sigma \models P_t * [1 = 0] * \text{ own}(x) * \text{ own}(y) \land (x = y), \) we know there exists \( \Sigma_1 \) such that \( \Sigma = \Sigma_1 \uplus \{ 1 \sim 0 \} \) and

\[
(\sigma, \Sigma_1) \models P_t * \text{ own}(x) * \text{ own}(y) \land (x = y)
\]

From \( D, R, G \models \{ P \} \Pi : \Gamma, \) we know: there exist three well-founded metrics \( u, M \) and \( M \) and a set \( \xi \in \mathcal{P}(\text{ThrillD}) \) such that

\[
\text{D, R, G } \models (C, \sigma) \subseteq ((C); \text{ return } E, \Sigma_1) \circ (u, M, M) \psi_{\xi} (P * \text{ own}(x) * \text{ own}(y)).
\]

We want to prove: there exist two well-founded metric \( \mathcal{H} \) and \( M \) and a set \( \xi \in \mathcal{P}(\text{ThrillD}) \) such that
Suppose
If For any
By co-induction. Below we use

\[
\begin{align*}
\text{wr}_1((C)) & \overset{\text{def}}{=} \text{wr}_1(\text{return } \Xi) \overset{\text{def}}{=} \\
\text{while ( } u_1 \geq 0 ) \{ & \text{lock } l;\\
& \text{unlock } l;\\
& u_1 := \text{rand_less_than}(u_1);\\
& (C);\\
\} \\
\text{wr}_1'(((C))) & \overset{\text{def}}{=} \text{wr}_1'(\text{return } \Xi) \overset{\text{def}}{=} \\
& \text{lock } l;\\
& \text{unlock } l;\\
& u_1 := \text{rand_less_than}(u_1);\\
& \text{while ( } u_1 \geq 0 ) \{\\
& \text{lock } l;\\
& \text{unlock } l;\\
& u_1 := \text{rand_less_than}(u_1);\\
& (C);\\
\} \\
\end{align*}
\]

Figure 22. Useful notations for the abstract code wrapper.

\[\mathcal{D}, R \models [1 = 0], G \models [1 = 0] \models \mathcal{L}^{\mathcal{G}}_1 (C, \sigma) \preceq (\text{wr}_1((C)); \text{return } \Xi), \Sigma) \circ ([M, M] \downarrow_{\xi} (P \ast (1 = 0) \ast \text{own}(x) \ast \text{own}(y)).\]

Fig 22 gives some useful notations for the abstract code wrapper. We only need to prove the following:

1. If \(\mathcal{D}, R, G \models_1 (C, \sigma) \preceq (\text{return } \Xi, \Sigma_1) \circ (u, M, M) \downarrow_{\xi} Q\)
   and \(\Sigma = \Sigma_1 \cup \{1 \sim 0, u_1 \sim u_1, u_2 \sim u_2\}\) and \(0 \leq u \leq u_1\) and \(0 \leq u \leq u_2\),
   then \(\mathcal{D}, R \ast [1 = 0], G \ast [1 = 0] \models_1 \mathcal{L}^{\mathcal{G}}_1 (C, \sigma) \preceq_1 (\text{wr}_1((C)); \text{return } \Xi), \Sigma) \circ ([M, M] \downarrow_{\xi} (Q \ast (1 = 0)).\)

2. If \(\mathcal{D}, R, G \models_1 (C, \sigma) \preceq (\text{return } \Xi, \Sigma_1) \circ (u, M, M) \downarrow_{\xi} Q\)
   and \(\Sigma = \Sigma_1 \cup \{1 \sim 0, u_1 \sim u_1, u_2 \sim u_2\}\) and \(0 \leq u \leq u_2\),
   then \(\mathcal{D}, R \ast [1 = 0], G \ast [1 = 0] \models_1 (C, \sigma) \preceq_1 (\text{wr}_1'(\text{return } \Xi), \Sigma) \circ ([M, M] \downarrow_{\xi} (Q \ast (1 = 0)).\)

By co-induction. Below we use \(\hat{C}\) for the abstract code in the above two goals.

1. Suppose \(\sigma = (s, h)\). Then \(\xi \subseteq s(\text{TIDS})\) and \(t \not\in \xi\).
   For any \(t' \in \xi\), we have \((\sigma, i) \models \text{Enabled}(\mathcal{D}_{t'}) \ast \text{true}.\)
   \textbf{Proof}: Immediate.

2. If \(C = E_1 \text{return } E\), then for any \(\Sigma_F\) such that \(\Sigma \cup \Sigma_F\), there exist \(\Xi\) and \(\Sigma'\) such that
   \(\text{(a) } (C, \Sigma \cup \Sigma_F) \rightarrow^* \text{return } \Xi, \Sigma' \cup \Sigma_F\), and
   \(\text{(b) } (\sigma, \Sigma') \models Q_1 \ast (1 = 0) \ast \text{true}_{\sigma, s} = [E_1]_{\Sigma', s}, \text{and}
   \(\text{(c) } ([\sigma, \Sigma), (\sigma, \Sigma')] ) \models Q_1 \ast [1 = 0] \ast \text{true}.\)
   \textbf{Proof}: Immediate from the premise.

3. For any \(\sigma_F, (C, \sigma \uplus \sigma_F) \rightarrow^* \text{abort}.\)
   \textbf{Proof}: Immediate.

4. For any \(C', \sigma', \sigma_F\) and \(\Sigma_F\), if \((C, \sigma \uplus \sigma_F) \rightarrow^* (C', \sigma'')\) and \(\Sigma \cup \Sigma_F\), then there exist \(\sigma', n, C', \Sigma', k, M', M'\) and \(\xi'\) such that
   \(\text{(a) } \sigma'' = \sigma' \uplus \sigma_F\), and
From (D1), we know: For 1, from (E1), we know: either $u < k$, $u = u'$ and $k = 0$ and $M' < M$, or $u < u$ and $k = 0$ and $M' = M$ and $\xi \neq 0$ and $\xi \subseteq \xi'$; and

From (B1), by the co-induction hypothesis, we know

$$\Delta_1(t, \Sigma, (\sigma', \Sigma')) \models G_t \models [1 = 0] \models \text{True}.$$ 

Thus we know

$$\Delta_1(t, \Sigma, (\sigma', \Sigma')) \models G_t \models [1 = 0] \models \text{True}.$$ 

Also we know

$$\Delta_1(t, \Sigma, (\sigma', \Sigma')) \models G_t \models [1 = 0] \models \text{True}.$$
From (B2), by the operational semantics, we know \( C' = \text{return}_1 \). Let

\[ C' = \text{return}_1 \] \( \) and \( u_2' = u' \) and \( \Sigma' = \Sigma_1 \uplus \{ 1 \leadsto 0, u_1 \leadsto u_1, u_2 \leadsto u_2 \} \).

From (C2), by the co-induction hypothesis, we know

\[ D', R * [1 = 0], G * [1 = 0] \models _{\text{co-ind}} (C', \sigma') \models (C', \Sigma') \circ (M', M') \downarrow_{\Sigma'} (Q * (1 = 0)) \).

From (D2), we know

\[ ((\sigma, \Sigma), (\sigma', \Sigma'), k) \models G_i * [1 = 0] * \text{True} \].

- If \( k > 0 \), from (E2), we know: \( u' < u \). Thus \( u_2' < u_2 \). Let

\[ C'' = \text{return}_1 \].

Thus we know

\[ \Delta_1 (t, \Sigma) = \Sigma_1 \uplus \{ 1 \leadsto t, u_1 \leadsto u_1, u_2 \leadsto u_2 \} \].

Also we know

\[ \tau_1 (C', \Sigma_1) \downarrow t (C', \Sigma_1) \uplus \Sigma_1 \].

(5) For any \( k, \sigma' \) and \( \Sigma' \), if \( ((\sigma, \Sigma), (\sigma', \Sigma'), k) \models \text{Enabled}(\mathcal{D}_1) \ast \text{True} \), then there exist \( M', M', \xi_d \) and \( \xi' \) such that

(a) \( D, R * [1 = 0], G * [1 = 0] \models _{\text{co-ind}} (C, \sigma') \models (C, \Sigma') \circ (M', M') \downarrow_{\Sigma'} (Q * (1 = 0)) \); and

(b) \( \xi_d = \{ t' \mid (t' \in \xi) \land ((\sigma, \Sigma), (\sigma', \Sigma')) \models (\mathcal{D}_1) \ast \text{Id} \} \) and \( (k = 0 \Rightarrow \xi \subseteq \xi') \), and

(c) if \( k > 0 \), then for any \( t' \neq t \) and \( \Sigma_1 \uplus \Sigma \) we have \( (C, \Delta_1 (\Sigma_1) \uplus \Sigma_1) \models (\mathcal{D}_1) \ast \text{True} \); otherwise, \( M' < M \), or \( M' = M \) and \( \xi_d = \emptyset \); and

(d) if \( k = 0 \) and \( (\sigma, \Sigma) \models \text{Enabled}(\mathcal{D}_1) \ast \text{True} \), then \( M' \leq M \).

**Proof:** Since \( \{ 1, u_1, u_2 \} \cap f_i (R) = \emptyset \), we know there exists \( \Sigma_1 \) such that

\[ ((\sigma, \Sigma_1), (\sigma', \Sigma_1), k) \models R_i \ast \text{Id} \) and \( \Sigma' = \Sigma_1 \uplus \{ 1 \leadsto 0, u_1 \leadsto u_1, u_2 \leadsto u_2 \} \).

Thus we know

\[ \Delta_1 (t', \Sigma) = \Sigma_1 \uplus \{ 1 \leadsto t', u_1 \leadsto u_1, u_2 \leadsto u_2 \} \].

Thus we know

\[ (\Delta_1 (t', \Sigma) \uplus \Sigma_1) \models (\text{return}_1 (C)); \text{return}_1 (C), \Delta_1 (t', \Sigma) \uplus \Sigma_1) \models _{\text{co-ind}} (\text{return}_1 (C)); \text{return}_1 (C), \Delta_1 (t', \Sigma) \uplus \Sigma_1) \]

and

\[ (\text{return}_1 (C), \Delta_1 (t', \Sigma) \uplus \Sigma_1) \models _{\text{co-ind}} (\text{return}_1 (C), \Delta_1 (t', \Sigma) \uplus \Sigma_1) \].

By the co-induction hypothesis, we can prove the goals.

Thus we are done. \( \square \)
B.4 Core Proofs: From Common Simulation to Contextual Refinement

Definition 20 (Simulation for Thread). \( D, R, G \models_{\Delta} \{ P \} (\Pi, C) \models (\Pi', C) \{ Q \} \) iff, for any \( \sigma, \epsilon, \) and \( \Sigma \), if \( (\sigma, \Sigma) \models P \), there exist two well-founded metrics \( \mathcal{M} \) and \( \mathcal{M}' \) and a set \( \epsilon \in \mathcal{P}(\text{ThreadD}) \) such that

\[
D, R, G \models_{\Delta} (\Pi, C, (\sigma, \epsilon, 0)) \models (\Pi', C, (\sigma, \Sigma, \delta)) \circ (\mathcal{M}, \mathcal{M}') \models_{\epsilon} Q.
\]

(See Sec. B.4.1 for their definitions.) By Lemma 21 we know: for any \( n, C_1, \ldots, C_n \),

\[
\{ \lambda_n P_i \} (\text{let } C_1 \| \ldots \| C_n) \models (\text{let } C' \in C_1 \| \ldots \| C_n).
\]

(Here the simulation \( \models \) for the whole programs is defined in Definition 24.) By the following Lemma 22 we know

\[
\{ \lambda_n P_i \} (\text{let } C_1 \| \ldots \| C_n) \subseteq (\text{let } C' \in C_1 \| \ldots \| C_n).
\]

(Here the fair refinement \( \subseteq \) is defined in Definition 23.) Thus we get: \( \Pi \sqsubseteq \Pi' \).

\[\square\]

B.4.1 Lifting to Simulation for Client Threads

Definition 20 (Simulation for Thread). \( D, R, G \models_{\Delta} \{ P \} (\Pi, C) \models (\Pi', C) \{ Q \} \) iff, for any \( \sigma, \epsilon, \) and \( \Sigma \), if \( (\sigma, \Sigma) \models P \), there exist two well-founded metrics \( \mathcal{M} \) and \( \mathcal{M}' \) and a set \( \epsilon \in \mathcal{P}(\text{ThreadD}) \) such that

\[
D, R, G \models_{\Delta} (\Pi, C, (\sigma, \epsilon, 0)) \models (\Pi', C, (\sigma, \Sigma, \delta)) \circ (\mathcal{M}, \mathcal{M}') \models_{\epsilon} Q.
\]

(See Sec. B.4.1 for their definitions.) By Lemma 21 we know: for any \( n, C_1, \ldots, C_n \),

\[
\{ \lambda_n P_i \} (\text{let } C_1 \| \ldots \| C_n) \models (\text{let } C' \in C_1 \| \ldots \| C_n).
\]

(Here the simulation \( \models \) for the whole programs is defined in Definition 24.) By the following Lemma 22 we know

\[
\{ \lambda_n P_i \} (\text{let } C_1 \| \ldots \| C_n) \subseteq (\text{let } C' \in C_1 \| \ldots \| C_n).
\]

(Here the fair refinement \( \subseteq \) is defined in Definition 23.) Thus we get: \( \Pi \sqsubseteq \Pi' \).

\[\square\]
B.4.2 Parallel Compositionality

**Lemma 22** (Parallel Compositionality).

If there exists $R, G, D, P$ and $\Delta$ such that the following hold for any $t \in \{1..n\}$:

1. $D, R, G \models_\Delta \{P\} \iff (\Pi, C_t) \subseteq (\Pi', C_t) \{P\}$.
2. $\forall t', t \neq t' \implies G_t \implies R_{t'}$. wffAct($R, D$), $P \implies \neg\text{Enabled}(D), P \lor \text{Enabled}(D) \implies I, I \triangleright I, R, G$.

then $(\bigwedge_t \Sigma_t) \{C_t \mid \ldots \{C_n\} \subseteq \{\Pi' \in C_t \mid \ldots \{C_n\}\}$.

**Proof.** For any $(\sigma, \Sigma, \sigma) \models (\bigwedge_t \Sigma_t)$, from the premises and by sequential compositionality, we can prove: there exist $M_1, \ldots, M_n, v_1, \ldots, v_n, \xi_1, \ldots, \xi_n$ such that the following holds for any $t \in \{1..n\}$:

\[
D, R, G \models_\Delta \{ \Pi, (C_t; \text{end}), (\sigma, \sigma, \circ) \} \subseteq \{ \Pi', (C_t; \text{end}), (\sigma, \Sigma, \circ) \} \circ (\Pi, M_t) \downarrow \xi_t, P .
\]

We want to show that there exist $\mathcal{M}$ and $\zeta$ such that

\[
\left( \text{let } \Pi \in (C_t; \text{end}) \mid \ldots \{C_n; \text{end}\}, (\sigma, \Sigma, \circ) \right) \subseteq \left( \text{let } \Pi' \in (C_t; \text{end}) \mid \ldots \{C_n; \text{end}\}, (\sigma, \Sigma, \circ) \right) \circ (\mathcal{M}, \zeta).
\]

We generalize the result and prove the following (B.3):

If $(\sigma, \Sigma) \models I$ and the following holds for any $t \in \{1..n\}$:

\[
D, R, G \models_\Delta \{ \Pi, C_t, (\sigma, \sigma \uplus \sigma_t, \kappa) \} \subseteq \{ \Pi', C_t, (\sigma, \Sigma \uplus \Sigma_t, \kappa) \} \circ (\Pi, M_t) \downarrow \xi_t, P ,
\]

then

\[
\left( \text{let } \Pi \in C_t \mid \ldots \{C_n\}, (\sigma, \sigma \uplus (\Sigma, \kappa_t)) \right) \subseteq \left( \text{let } \Pi' \in C_t \mid \ldots \{C_n\}, (\sigma, \Sigma \uplus (\Sigma_t, \kappa)) \right) \circ (\mathcal{M}, \zeta).
\]

Here for any $t \in \{1..n\}$, $K(t) = \kappa_t$ and $\mathcal{K}(t) = \kappa_t$, and the functions $\mathcal{M}$ and $\zeta$ are defined as follows.

- $\text{dom}(\mathcal{M}) = \text{activeThrs}(\text{let } \Pi \in C_t \mid \ldots \{C_n\}) = \{ t \mid (C_t \neq \text{skip}) \}$.
- $\text{dom}(\zeta) = \text{activeThrs}(\text{let } \Pi \in C_t \mid \ldots \{C_n\}) = \{ t \mid (C_t \neq \text{skip}) \}$. 

The order $(M', M') < (M, M)$ is defined as a dictionary order:

\[
(M', M') < (M, M) \text{ iff } (M' < \Pi') \lor (M' = \Pi') \land (M' < M)
\]

\[
M' < M \text{ iff } \exists t. (M'(t) < M(t)) \land (M'(t') \leq M(t'))
\]

\[
M' < M \text{ iff } \forall t' \in \text{dom}(M'). (M'(t') \leq M(t'))
\]

Clearly $\mathcal{M}'(t) < \mathcal{M}(t)$ is a well-founded order. (B.3)

By co-induction. Let $W \equiv (\text{let } \Pi \in C_t \mid \ldots \{C_n\}), W' \equiv (\text{let } \Pi' \in C_t \mid \ldots \{C_n\}), S \equiv (\sigma, \sigma \uplus (\Sigma, \kappa_t)), S' \equiv (\sigma, \Sigma \uplus (\Sigma_t, \kappa))$.

Suppose $\sigma = (s, h)$. Then $s(\text{TIDS}) = \{1..n\}$.

1. $\text{dom}(\mathcal{M}) = \text{dom}(\zeta) = \text{activeThrs}(W) = \text{activeThrs}(W), \text{ and } \forall t \in \text{dom}(\zeta), \zeta(t) \subseteq \{ \text{dom}(\zeta) \} \{t\}$.

   **Proof.** For any $t \in \{1..n\}$, from the premise, we know: $(C_t \neq \text{skip}) \implies (C_t \neq \text{skip})$. Thus $\text{activeThrs}(\text{let } \Pi \in C_t \mid \ldots \{C_n\}) = \text{activeThrs}(\text{let } \Pi' \in C_t \mid \ldots \{C_n\})$.

   Also, $\zeta(t) \subseteq s(\text{TIDS}) = \{1..n\}$. And for any $t' \in \zeta(t)$, we have $t' \neq t$ and $(\sigma, \Sigma) \models \text{Enabled}(D_{t'}) \iff \text{true}$. Then we only need to prove $(C_{t'} \neq \text{skip})$. Suppose $(\sigma, \Sigma) \models \text{Enabled}(D_{t'})$, then we have $(\sigma, \Sigma) \models \text{Enabled}(D_{t'})$. Thus $P_t' \land \text{Enabled}(D_{t'}) \implies \text{false}$, we get $(\sigma, \Sigma) \models \text{false}$, which is impossible. Thus $(C_{t'} \neq \text{skip})$.

2. If $(W, S) \implies (\text{skip}, S')$, then there exist $T$ and $S'$ such that getObsv$(T) = e$ and $(W, S) \implies (\text{skip}, S')$.

   **Proof.** By the operational semantics we know: for any $t \in \{1..n\}$, we have $C_t = \text{skip}$. By $D, R, G \models_\Delta (\Pi, C_t, (\sigma, \sigma \uplus \sigma_t, \kappa_t)) \subseteq (\Pi', C_t, (\sigma, \Sigma \uplus \Sigma_t, \kappa)) \circ (\Pi, M_t) \downarrow \xi_t, P$, we have $C_t = \text{skip}$. Thus we have $(W, S) \implies (\text{skip}, S)$.

3. If $(W, S) \implies (t', \text{abort})$, then there exist $T$ and $S' \iff e = \text{getObsv}(T)$ and $(W, S) \implies (t', \text{abort})$.

   **Proof.** By the operational semantics we know: there exists $t \in \{1..n\}$ such that

\[
(C_t, (\sigma, \sigma \uplus (\Sigma, \kappa_t))) \implies (t, \text{abort}).
\]

By $D, R, G \models_\Delta (\Pi, C_t, (\sigma, \sigma \uplus \sigma_t, \kappa_t)) \subseteq (\Pi', C_t, (\sigma, \Sigma \uplus \Sigma_t, \kappa)) \circ (\Pi, M_t) \downarrow \xi_t, P$, we know $e = (t, \text{abort})$ and there exists $T$ such that $e = \text{getObsv}(T)$ and

\[
(C_t, (\sigma, \Sigma \uplus (\Sigma_t, \kappa_t))) \implies_1 (t, \text{abort}).
\]

Thus $(W, S) \implies (\text{abort}, S)$.

4. If $(W, (\sigma, \sigma \uplus (\Sigma, \kappa_t))) \implies (W', (\sigma', \sigma''', \kappa'))$, then there exist $T$, $W'$, $S'$, $\mathcal{M}'$ and $\zeta'$ such that all the following hold:

   a. $(W', (\sigma, \Sigma \uplus (\Sigma_t, \kappa))) \implies (W', S')$;
   b. $t = \text{tid}(e)$. getObsv$(e) = \text{getObsv}(T)$, $(e = (t, \text{term})) \implies (e = \text{last}(T))$;
   c. $(W', (\sigma', \sigma'''', \kappa')) \subseteq (W', S') \circ (\mathcal{M}', \zeta')$;
Proof: By the operational semantics, we know there exist \( t, C'_t \) and \( \kappa'_t \) such that

\[
W' = (\text{let } \Pi \ \text{in } C_1 \parallel \ldots \parallel C_n), \quad t = \text{tid}(e),
\]

\[
(C_t, (\sigma, \sigma \uplus \Sigma, \kappa), k_t) \xrightarrow{t, \Pi} (C'_t, (\sigma', \Sigma', \kappa'), k'_t) \text{ and } \kappa' = \kappa \cup \{t \rightsquigarrow \kappa'_t\}.
\]

By \( D, R, G \models \Delta (\Pi, C_t, (\sigma, \sigma \uplus \Sigma, \kappa)) \supseteq (\Pi', C'_t, (\sigma', \Sigma', \kappa')) \circ (\Pi, M_t) \parallel \psi_e P, \) we know

(A) For any \( t' \in \xi_t \), we have \( t' \neq t \) and \( (\sigma \uplus \sigma_t, \Sigma \uplus \Sigma_t) \models \text{Enabled}(D_{e'}) \times \text{true}. \)

And there exist \( \sigma''', \sigma'' \in \Sigma, \Sigma'' \subseteq \Sigma \) such that

\[
(\sigma'') \uplus \sigma'' \subseteq \sigma', \quad (\sigma', \Sigma'') \uplus \Sigma'' \subseteq (\Pi, M_t) \parallel \psi_e P, \text{ and}
\]

(B) \( \sigma'''' = \sigma'' \uplus \Sigma'' \subseteq \Sigma \) and

\[
(\Pi, C''_t, (\sigma, \Sigma \uplus \Sigma_t, \kappa)) \xrightarrow{t''''} (C'_t, (\sigma', \Sigma', \kappa')) \text{ and } \kappa' = \kappa \uplus \{t'''' \rightsquigarrow \kappa'_{t''''}\},
\]

is such that

\[
(\sigma''') \uplus \sigma'''' \subseteq \Sigma \text{ and } (\sigma', \Sigma'') \uplus \Sigma''' \subseteq (\Pi, M_t) \parallel \psi_e P, \text{ and}
\]

(C) \( (\Pi, C''_t, (\sigma, \Sigma \uplus \Sigma_t, \kappa)) \xrightarrow{t''''} (C'_t, (\sigma', \Sigma', \kappa')) \text{ and } \kappa' = \kappa \uplus \{t'''' \rightsquigarrow \kappa'_{t''''}\},
\]

is such that

\[
(\sigma'', \Sigma'', \kappa) \subseteq (\Pi, C''_t, (\sigma, \Sigma \uplus \Sigma_t, \kappa)) \text{ and } (\sigma', \Sigma', \kappa') \subseteq (\Pi, C'_t, (\sigma', \Sigma', \kappa') \cup \{t'''' \rightsquigarrow \kappa'_{t''''}\}, k'_t)}.
\]

For any \( t' \neq t \), since \( G_t \Rightarrow R_{t'} \), we have \( ((\sigma, \Sigma), (\sigma', \Sigma')) \models \text{Enabled}(D_{t'}) \times \text{true}. \)

Thus

\[
((\sigma \uplus \sigma_t, \Sigma \uplus \Sigma_t), (\sigma' \uplus \sigma_t, \Sigma' \uplus \Sigma_t), k) \models R_{t'} \text{ and ld}.
\]

By \( D, R, G \models \Delta (\Pi, C_t, (\sigma, \sigma \uplus \sigma_t, \kappa)) \supseteq (\Pi', C'_t, (\sigma', \Sigma' \uplus \Sigma_t, \kappa')), \) we know

(I) For any \( t'''' \in \xi_t \), we have \( (\sigma''') \neq t'''' \neq t \) and \( (\sigma \uplus \sigma_t, \Sigma \uplus \Sigma_t) \models \text{Enabled}(D_{t''''}) \times \text{true}. \)

And there exist \( \psi_{t''''}, M'_{t''''}, \xi_{t''''}, \kappa'_{t''''} \) such that

\[
(\Pi, C'_t \cup C''_t, (\sigma, \Sigma \uplus \Sigma_t, \kappa)) \models (\Pi', C'_t, (\sigma', \Sigma' \uplus \Sigma_t, \kappa'), (\Pi, M_t) \parallel \psi_{t''''} P, \text{ and}
\]

(J) \( \xi_{t''''} = \{t'''' \in \xi_t \cap (\Pi, C'_t \cup C''_t, (\sigma, \Sigma \uplus \Sigma_t, \kappa)) = (\Pi, D_{t''''}) \cup \text{ld}\} \) and \( k = 0 \Rightarrow \xi_{t''''} = \xi_{t''''} \text{ and }\)

(L) if \( k > 0 \), then there exists \( \Sigma''' \) such that \( (\Pi', C'_t, (\sigma, \Sigma' \uplus \Sigma_t, \kappa')) \xrightarrow{t''''} (\Pi', C'_t, (\sigma, \Sigma' \uplus \Sigma_t, \kappa')) \text{ and } (\sigma', \Sigma' \uplus \Sigma_t, \kappa') \uplus \Sigma'''' \subseteq (\Pi, M_t) \parallel \psi_{t''''} P, \text{ and}
\]

(M) if \( k = 0 \) and \( (\sigma \uplus \sigma_t, \Sigma \uplus \Sigma_t) \models \text{Enabled}(D_{t''''}) \times \text{true}, \text{ then } M'_{t''''} \leq M_t. \)

From (C), we know

\[
(C_t, (\sigma, \Sigma \uplus \Sigma_t, \kappa_t)) \xrightarrow{t, \Pi} (C'_t, (\sigma', \Sigma' \uplus \Sigma_t, \kappa'_t), k'_t).
\]

Let \( \mathcal{W}' = (\text{let } \Pi' \ \text{in } C_1 \parallel \ldots \parallel C_n \parallel S') \)

\[
(\sigma', \Sigma' \uplus \Sigma_t \cup ((\Pi, M_t) \parallel \psi_t P, \text{ and}
\]

Also, for any \( t' \neq t \), we have: either \( t' \in \text{tidset}(T) \), or \( \psi_{t''''} = \psi_t P \) and \( \xi_{t''''} = \emptyset \).

From (G), we know: either \( t \in \text{tidset}(T) \), or \( \psi_{t''''} = \psi_t P \) and \( \xi_{t''''} = \emptyset \) and \( \xi_{t''''} = \emptyset \).

Besides, we have

\[
k > 0 \Rightarrow (t \in \text{tidset}(T)) \wedge (t' \in \text{tidset}(T)).
\]

Define the functions \( \mathcal{M}' \) and \( \mathcal{M}_t' \) as follows.

- \( \text{dom}(\mathcal{M}') = \text{dom}(\mathcal{M}) = \text{activeThds}(W') \).
- For any \( t \in \text{dom}(\mathcal{M}') \), we have \( \mathcal{M}'(t) = (\Psi_t, \{t' \rightsquigarrow M'_t \mid t' \in \xi'_t\}) \) and \( \mathcal{M}_t'(t) = \xi'_t \).

Then by the co-induction hypothesis, we know

\[
(\mathcal{W}', (\sigma', \Sigma', \kappa'_t)) \subseteq (\mathcal{W}', S') \cup (\mathcal{M}', \mathcal{M}_t') \rightarrow (\mathcal{M}', \mathcal{M}_t').
\]

For the thread \( t \), suppose \( t \notin \text{tidset}(T) \), then \( k = 0 \). Then,
• If $\nu'_t < \nu_t$, then $\mathcal{M}'(t) < \mathcal{M}(t)$.
• If $\nu'_t = \nu_t$ and $\xi_t \neq \emptyset$ and $\xi_t \subseteq \xi'_t$, we know $\zeta(t) \neq \emptyset$ and $\zeta(t) \subseteq \zeta'(t)$. We only need to show the following (B.4):

$$\{ t' \sim M'_t \mid t' \in \xi'_t \} \subseteq \{ t' \sim M_t \mid t' \in \xi_t \}$$

(B.4)

From (A), since $\forall t'$. Enabled($D_{t'}$) $\Rightarrow I$ and Precise($I$), we know: for any $t' \in \xi_t$, $t' \neq t$ and $(\sigma, \Sigma) \models$ Enabled($D_{t'}$).

Thus $(\sigma \cup \sigma', \Sigma \cup \Sigma_t) \models$ Enabled($D_{t'}$) $\Rightarrow$ true. From (M), we have $M'_t \leq M_t$. Thus (B.4) holds.

For any $t' \in dom(\mathcal{M}\setminus\{t\})$, suppose $t' \not\in \text{tidset}(T)$, then $k = 0$. Then,

• If $\nu'_t < \nu_t$, then $\mathcal{M}'(t') < \mathcal{M}(t')$.
• If $\nu'_t = \nu_t$, $\xi_t = \emptyset$ and $t \not\in \xi_t$, from (K), we know $\xi_t \subseteq \xi'_t$. Thus $t \not\in \zeta'(t)$ and $\zeta(t) \subseteq \zeta'(t')$. We only need to show the following (B.5):

$$\{ t'' \sim M'_t \mid t'' \in \xi'_t \} \subseteq \{ t'' \sim M_t \mid t'' \in \xi_t \}$$

(B.5)

From (I), we know: for any $t'' \in \xi_t$, $t'' \neq t'$ and $(\sigma, \Sigma) \models$ Enabled($D_{t''}$).

Thus $(\sigma \cup \sigma', \Sigma \cup \Sigma_t) \models$ Enabled($D_{t''}$) $\Rightarrow$ true. From (M), we have $M'_t \leq M_t$. Thus (B.5) holds.

• If $\nu'_t = \nu_t$, $\xi_t = \emptyset$ and $t \in \xi_t$, since $\underline{wffAct}(R, D)$ and $k = 0$, we know $((\sigma, \Sigma), (\sigma', \Sigma')) \models \langle [D_t] \rangle$.

From (H), we know $M'_t < M_t$.

We only need to show the following (B.6):

$$\{ t'' \sim M'_t \mid t'' \in \xi'_t \} < \{ t'' \sim M_t \mid t'' \in \xi_t \}$$

(B.6)

For any $t'' \in \xi_t \setminus \{t\}$, from (I), we know:

$$t'' \neq t' \text{ and } (\sigma, \Sigma) \models \text{Enabled}(D_{t''}).$$

Thus $(\sigma \cup \sigma', \Sigma \cup \Sigma_t) \models \text{Enabled}(D_{t''}) \Rightarrow \text{true}$. From (M), we have $M'_t \leq M_t$. Thus (B.6) holds.

Thus we are done.

\[\square\]

### B.4.3 Simulation for Whole Programs and Fair Refinement

**Definition 23** (Fair refinement). $\{P\}W \subseteq \mathcal{W}$ iff

$$\forall \sigma, \Sigma, (\sigma, \Sigma) \models P \implies \mathcal{O}_w([W, (\sigma, \Sigma, \emptyset)]) \subseteq \mathcal{O}_w([W, (\sigma, \Sigma, \emptyset)])$$

**Definition 24** (Simulation for the whole program). $\{P\}W \preceq \mathcal{W}$ iff, for any $\sigma, \sigma'$ and $\Sigma$, if $(\sigma, \Sigma) \models P$, there exist $\mathcal{M} \in \text{ThrID} \rightarrow \text{Metric}$ and $\zeta \in \text{ThrID} \rightarrow \mathcal{P}(\text{ThrID})$ such that

$$([W], (\sigma, \Sigma, \emptyset)) \preceq ([W], (\sigma, \Sigma, \emptyset)) \circ \mathcal{M}, \zeta.$$

Here $(W, S) \prec (W, S) \circ \mathcal{M}, \zeta$ is co-inductively defined as follows.

Whenever $(W, S) \preceq (W, S) \circ \mathcal{M}, \zeta$ holds, then the following hold:

1. $\text{dom}(\mathcal{M}) = \text{dom}(\zeta) = \text{activeThrds}(W) = \text{activeThrds}(\mathcal{W})$, and $\forall t \in \text{dom}(\zeta), \zeta(t) \subseteq (\text{dom}(\zeta) \setminus \{t\})$.
2. $\text{If } (W, S) \rightarrow (\text{skip}, S'), \text{ then there exist } T$ and $S'$ such that get OBS($T$) = $\epsilon$ and $(W, S) \longrightarrow^+ (\text{skip}, S')$.
3. $\text{If } (W, S) \rightarrow \text{abort}, \text{ then there exist } T$ and $S$ such that $e = (t, \text{clt}. \text{abort})$, $e \models \text{get OBS}(T)$ and $(W, S) \rightarrow^+ \text{abort}$.
4. $\text{If } (W, S) \rightarrow^+ (W', S'), \text{ then there exist } T, \text{ activeThrds}(T), \mathcal{M}, \text{ and } \zeta$ such that the following hold:

   a. $(W, S) \rightarrow^+(W', S')$.
   b. $t = \text{tid}(e)$, get OBS($e$) = get OBS($T$), $(e = (t, \text{term})) \Rightarrow (e = \text{last}(T))$.
   c. $(W', S') \preceq ([W', S'], \mathcal{M}, \zeta)$
   d. either $t \in \text{tidset}(T)$,
      or $\mathcal{M}'(t) < \mathcal{M}(t)$,
      or $\mathcal{M}'(t) = \mathcal{M}(t)$ and $\zeta(t) \neq \emptyset$ and $\zeta(t) \subseteq \zeta'(t)$.
   e. for any $t' \in \text{dom}(\mathcal{M}) \setminus \{t\}$, we have:
      either $t' \in \text{tidset}(T)$,
      or $\mathcal{M}'(t') < \mathcal{M}(t')$,
      or $\mathcal{M}'(t') = \mathcal{M}(t')$ and $t \not\in \zeta'(t')$ and $\zeta(t') \subseteq \zeta'(t')$.

**Lemma 25** (Simulation for whole program ensures fair refinement).

If $\{P\}W \preceq \mathcal{W}$ and $|W| = |\mathcal{W}|$, then $\{P\}W \subseteq \mathcal{W}$.

To prove Lemma 25, we introduce the notion of scheduler $\Sigma$, which is similar to the event trace $T$ but contains more information such that it can uniquely determine an execution. That is, each step is deterministic given the code, the state, and the scheduler. Note that the event trace $T$ is not sufficient to determine a unique execution if we allow non-deterministic instructions such as $x := r$ and $\emptyset$.
{(let \( \Pi \) in \( C_1 \| \ldots \| C_n \)) \( \overset{\text{def}}{=} \) \( C\)}

\[ \text{activeThrs}(W) \overset{\text{def}}{=} \{ t \mid \exists C. (W|_t = C) \land (C \neq \text{skip}) \} \]

\[ \text{tidset}(T) \overset{\text{def}}{=} \{ \emptyset \} \cup \text{tidset}(T') \]

\[ \text{if } T = \epsilon \]

\[ \text{if } T = \epsilon ; T' \]

**Figure 23.** Auxiliary definitions.

\[(\text{SchEvt}) \quad \iota ::= (t, f, n) | (t, \text{ret}, n) | (t, \text{obj}, \text{info}) | (t, \text{obj}, \text{abort}) | (t, \text{out}, n, \text{info}) | (t, \text{clt}, \text{info}) | (t, \text{clt}, \text{abort}) | (t, \text{term}) | (\text{spawn}, n)\]

\[(\text{Sched}) \quad \Xi ::= \epsilon | \iota :: \Xi \] (co-inductive)

Here \( \text{info} \) denotes the additional information that can uniquely determine a step. For instance, it could be the value written to \( x \) for the non-deterministic instruction \( x := \text{rand}(0) \). We can even record the initial and final states of the step if necessary.

We strengthen the labelled transition system \( (W, S) \rightarrow (W', S') \) in Fig. 5 to \( (W, S) \rightarrow^* (W', S') \). The definition of the strengthened transition system is similar to the original one and omitted here. Then \( (W, S) \rightarrow^* (W', S') \) represents a zero or multi-step execution of \( (W, S) \) that leads to \( (W', S') \) under the scheduler \( \Xi \). \( (W, S) \rightarrow^* \) represents an infinite execution with the infinite scheduler \( \Xi \). We use \( \epsilon = [\iota] \) to remove the additional information in \( \iota \), and \( T = [\Xi] \) to remove the additional information in each event in \( \Xi \). We also overload all the predicates over \( T \) to be defined over \( \Xi \). For instance, \( \text{get\_obsv}(\Xi) \) gives us an observable “scheduler” \( \Xi_o \), which is the subsequence of \( \Xi \) consisting of externally observable events only.

To facilitate the proofs, we give alternative definitions of \( \text{O}_\omega \). Fig. [24] shows the alternative co-inductive definitions for generating observable event traces of complete fair executions. We prove they are equivalent to the original definitions, as shown in the following lemmas. Their proofs are given later.

**Lemma 26.** \( (T_o \in \text{O}_\omega[U, S]) \iff (\exists \Xi, \Xi_o. (\Xi \models \text{O}_\omega[U, S, \Xi_o]) \land ([\Xi_o] = T_o)) \).

**Lemma 27.** \( (\exists \Xi. \Xi \models \text{O}_\omega[O, S, \Xi_o]) \iff \text{O}_\omega^p[(W, S), \Xi_o] \).

**Lemma 28.** If \( M \models \text{O}_\omega^p(W, S, \Xi_o) \), then \( \text{O}_\omega^p(W, S, \Xi_o) \).

We also define a simulation between whole programs under an explicit scheduler \( \Xi \) at the low level, \( \Xi \models (W, S) \preceq (W, S) \circ M \), as a bridge that relates the simulation \( (W, S) \preceq (W, S) \circ \Phi \) and \( \Phi \) in Definition 24 to the fair refinement.

**Definition 29** (Simulation for the whole program with fixed scheduling at the low level), \( \Xi \models (W, S) \preceq (W, S) \circ M \) is co-inductively defined as follows. Here \( M \in \text{ThrID} \rightarrow \text{Metric} \). Whenever \( \Xi \models (W, S) \preceq (W, S) \circ M \) holds, then the following hold:

1. \( \text{dom}(M) = \text{activeThrs}(W) = \text{activeThrs}(W) \).
2. If \( (W, S) \rightarrow (\text{skip}, S') \) and \( \Xi = \epsilon \), then there exist \( T \) and \( S' \) such that \( \text{get\_obsv}(T) = \epsilon \) and \( (W, S) \rightarrow^* (\text{skip}, S') \).
3. If \( (W, S) \rightarrow^* \text{abort} \) and \( \Xi = \iota :: \Xi \), then there exist \( t \) and \( S' \) such that \( |t| = |t, \text{clt}, \text{abort}|, |t| = \text{get\_obsv}(T) \) and \( (W, S) \rightarrow^* (\text{skip}, S') \).
4. If \( (W, S) \rightarrow (W', S') \) and \( \Xi = \iota :: \Xi \), then there exist \( t, T, W', S' \) and \( M' \) such that all the following hold:
   a. \( (W, S) \rightarrow^* (W', S') \);
   b. \( t = \text{tid}(|t|), \text{get\_obsv}(|t|) = \text{get\_obsv}(T), (|t| = \text{term}) \Rightarrow (|t| = \text{last}(T)) \);
   c. \( \Xi | (W', S') \preceq (W', S') \circ M' \);
   d. for any \( t' \in \text{dom}(M) \), either \( t' \in \text{tidset}(T) \), or \( M'(t') < M(t') \).

**Lemma 30.** For any \( \Xi, W, S, \Xi_o, W, S, \Phi \) and \( \xi \), if \( \Xi \models \text{O}_\omega^p(W, S, \Xi_o) \) and \( (W, S) \preceq (W, S) \circ \Phi \), then there exists \( M \) such that \( \Xi \models (W, S) \preceq (W, S) \circ M \).

**Lemma 31.** If \( \Xi \models \text{O}_\omega^p(W, S, \Xi_o) \) and \( \Xi \models (W, S) \preceq (W, S) \circ M \), then \( M \models \text{O}_\omega^p(W, S, \Xi_o) \).

The proofs of the above lemmas are given later. With the auxiliary simulation with an explicit scheduler and these useful lemmas, we can finish the proof of Lemma 25.
Co-inductive definitions for generating observable event traces of complete fair executions.

\[
(W, S) \xrightarrow{\tau} + \text{abort} \quad \delta_{\text{obs}}(\tau) = I_o
\]

\[
(W, S) \xrightarrow{\tau} + (W', S') \quad \delta_{\text{obs}}(\tau) = \epsilon \quad T' = C^{\text{obs}}_{\text{co}}(W', S', \epsilon) \quad \text{activeThrds}(W) \subseteq \text{tidset}(\tau)
\]

(a) with an explicit scheduler

\[
(W, S) \xrightarrow{\tau} + (W', S') \quad \delta_{\text{obs}}(\tau) = \epsilon \quad O^{\text{obs}}_{\text{co}}(W', S', I_o) \quad \text{activeThrds}(W) \subseteq \text{tidset}(\tau)
\]

(b) the scheduler is implicit

\[
(W, S) \xrightarrow{\tau} + (W', S') \quad \delta_{\text{obs}}(\tau) = \epsilon \quad O^{\text{obs}}_{\text{co}}(W', S', I_o) \quad \text{activeThrds}(W) \subseteq \text{tidset}(\tau)
\]

(c) with the metric mapping \( \mathcal{M} \in \text{ThrdID} \rightarrow \text{Metric} \)

Figure 24. Co-inductive definitions for generating observable event traces of complete fair executions.
Proof of Lemma 25. By Lemmas 26 and 27 we only need to show: for any \( \sigma_c, \sigma, \Sigma, i, \Xi \) and \( \Xi_o \), if \( (\sigma, \Sigma, i) \models P \) and \( \Xi \models O^\omega_{\Xi_o}(\lfloor W \rfloor, (\sigma, \sigma, \otimes), \Xi_o) \).

From \( \{P\}W \preceq W \), we know there exists \( \mathcal{M} \) and \( \zeta \) such that
\[
(\lfloor W \rfloor, (\sigma_c, \sigma, \otimes)) \preceq (\lfloor W \rfloor, (\sigma, \Sigma, \otimes)) \circ (\mathcal{M}, \zeta).
\]

By Lemma 30 we know there exists \( \mathcal{M} \) such that
\[
\Xi \models (\lfloor W \rfloor, (\sigma_c, \sigma, \otimes)) \preceq (\lfloor W \rfloor, (\sigma, \Sigma, \otimes)) \circ \mathcal{M}.
\]

Then by Lemma 31 we know
\[
\mathcal{M} \models O^\omega_{\Xi_o}(\lfloor W \rfloor, (\sigma_c, \Sigma, \otimes), \Xi_o).
\]

By Lemma 28 we get
\[
O^\omega_{\Xi_o}(\lfloor W \rfloor, (\sigma_c, \Sigma, \otimes), \Xi_o).
\]

Thus we are done.
Proof of Lemma 26. \( T_0 \in O_{\omega}[W,S] \) can be unfolded as follows.

\[
T_0 \in O_{\omega}[W,S] \iff \exists T. (([W], S) \xrightarrow{T} + \text{abort}) \land (T_0 = \text{get}\_\text{obsv}(T)) \\
\lor (([W], S) \xrightarrow{T} + (\text{skip}, \_)) \land (T_0 = \text{get}\_\text{obsv}(T)) \\
\lor (([W], S) \xrightarrow{T} \omega) \land (T_0 = \text{get}\_\text{obsv}(T)) \\
\land (\forall t \in \text{activeThrds}([W]). |([T_1]|) = \omega \lor \text{last}(T_1) = (t, \text{term}))
\]

We define \((\Sigma_0, \Sigma) \in O_{\omega}[W,S]\) as follows.

\[
(\Sigma_0, \Sigma) \in O_{\omega}[W,S] \iff ((W, S) \xrightarrow{\Sigma} + \text{abort}) \land (\Sigma_0 = \text{get}\_\text{obsv}(\Sigma)) \\
\lor ((W, S) \xrightarrow{\Sigma} + (\text{skip}, \_)) \land (\Sigma_0 = \text{get}\_\text{obsv}(\Sigma)) \\
\lor ((W, S) \xrightarrow{\Sigma} \omega) \land (\Sigma_0 = \text{get}\_\text{obsv}(\Sigma)) \\
\land (\forall t \in \text{activeThrds}(W). |([\Sigma]|) = \omega \lor \text{last}(\Sigma) = (t, \text{term}))
\]

Then \((T_0 \in O_{\omega}[W,S]) \iff (\exists \exists \Sigma_0, \Sigma. ((\Sigma_0, \Sigma) \in O_{\omega}([\text{W}]), S) \land (\Sigma_0 = T_0)).\) Below we prove

\[
(\exists \Sigma_0, \Sigma. ((\Sigma_0, \Sigma) \in O_{\omega}[W,S]) \iff (\forall t \in \text{activeThrds}(W). |([\Sigma]|) = \omega \lor \text{last}(\Sigma) = (t, \text{term}))
\]

\[\text{(B.7)}\]

\[\] • \(\iff\): We have two cases:

- \(|\Sigma| \neq \omega:\)
  - \text{Proofby induction over }|\Sigma|, \text{ and then by inversion over }\Sigma = O_{\omega}^{=}(W, S, \Sigma_0).

- \(|\Sigma| = \omega:\)
  - We prove \((W, S) \xrightarrow{\Sigma} \omega\), \(\Sigma_0 = \text{get}\_\text{obsv}(\Sigma)\) and \(\forall t \in \text{activeThrds}(W). |([\Sigma]|) = \omega \lor \text{last}(\Sigma) = (t, \text{term})\).
    - We prove \((W, S) \xrightarrow{\Sigma} \omega\) by co-induction.
    - We prove \(\Sigma_0 = \text{get}\_\text{obsv}(\Sigma)\) by co-induction.
    - Let \(\text{termThrds}(\Sigma) \overset{\Delta}{=} \{t \mid \text{last}(\Sigma) = (t, \text{term})\}\). For any \(t \in \text{activeThrds}(W) \setminus \text{termThrds}(\Sigma)\), we prove \(|([\Sigma]|) = \omega\) by co-induction.

- \(\iff\): By co-induction.

\[\] \[\]

Proof of Lemma 27 By co-induction.
Proof of Lemma 28: We want to prove the following:

\[ \forall W, S, \Upsilon_0, (\exists M. (M \models O^\omega_{\Upsilon_0}(W, S, \Upsilon_0))) \implies O^\omega_{\Upsilon_0}(W, S, \Upsilon_0) \]

By co-induction.

Co-induction Principle: \( \forall x. (\exists S. S \subseteq F(S) \land x \in S) \implies x \in \text{gfp } F \)

Figure 24 defines \( F \) and \( \text{gfp } F \) (i.e., \( O^\omega_{\Upsilon_0} \) at the middle part of the figure). Let

\[ S \overset{\text{def}}{=} \{(W, S, \Upsilon_0) \mid \exists M. (M \models O^\omega_{\Upsilon_0}(W, S, \Upsilon_0))\}. \]

So from the co-induction principle, we only need to prove:

\[ S \subseteq F(S), \text{ i.e., } \forall W, S, \Upsilon_0. (W, S, \Upsilon_0) \in S \implies (W, S, \Upsilon_0) \in F(S). \]

By unfolding \( S \) and by inversion over \( O^\omega_{\Upsilon_0} \), we have the following cases.

1. \( (W, S) \xrightarrow{\tau_y} \text{abort} \) and \( \text{get\_obs}(\Upsilon_0) = \Upsilon_0 \):
   From the definition of \( F \) (at the middle part in Figure 24), we know \( (W, S, \Upsilon_0) \in F(S) \).

2. \( (W, S) \xrightarrow{\tau_y} \text{skip} \) and \( \text{get\_obs}(\Upsilon_0) = \Upsilon_0 \):
   From the definition of \( F \) (at the middle part in Figure 24), we know \( (W, S, \Upsilon_0) \in F(S) \).

3. \( \Upsilon_0 = \epsilon, (W, S) \xrightarrow{\tau_x}^* (W_y, S_y), \text{get\_obs}(\Upsilon_x) = \epsilon, (M_y \models O^\omega_{\Upsilon_y}(W_y, S_y, \epsilon)) \),
   \( \text{dom}(M) = \text{active\_Thrs}(W) \neq \emptyset \) and \( \forall t \in \text{dom}(M) \text{ \mid tidset}(\Upsilon_x) \).
   We want to prove \( (W, S, \epsilon) \in F(S) \).
   We have two cases:
   (a) \( \text{active\_Thrs}(W) \nsubseteq \text{tidset}(\Upsilon_x) \):
      Since \( \text{active\_Thrs}(W) \neq \emptyset \), we know \( (W, S) \xrightarrow{\tau_x} (W_y, S_y) \).
      From \( (M_y \models O^\omega_{\Upsilon_y}(W_y, S_y, \epsilon)) \), we know \( (W_y, S_y, \epsilon) \in S \). From the definition of \( F \) (at the middle part in Figure 24), we know \( (W, S, \epsilon) \in F(S) \).
   (b) \( \text{dom}(M) \nsubseteq \text{tidset}(\Upsilon_x) \):
      By transfinite induction over the metric \( \text{dom}(M) \text{\mid tidset}(\Upsilon_x), M_y \).
      Transfinite Induction Principle: \( \forall M. (M', M < M \implies P(M')) \implies P(M) \implies \forall M. P(M) \)
   The order over the metrics, \( (\xi_2, M_2) < (\xi_1, M_1) \), is defined as follows.
   \[ (\xi_2, M_2) < (\xi_1, M_1) \iff (\xi_2 \subseteq \xi_1) \lor (\xi_2 = \xi_1 \land \forall t \in \xi_2, M_2(t) < M_1(t)) \]
   It is clear that \( (\xi_2, M_2) < (\xi_1, M_1) \) is a well-founded order.
   We view our goal as \( \forall M, \xi_x, \xi_y, P(M, \xi_x, \xi_y, M_y) \), which is reformulated as follows.
   \[ \forall M, \xi_x, \xi_y, W_y, S_y, (\xi_x, \xi_y) \nsubseteq (\xi_x, \xi_y) \land (\text{get\_obs}(\Upsilon_x) = \epsilon) \]
   \[ \land (M_y \models O^\omega_{\Upsilon_y}(W_y, S_y, \epsilon)) \land (\forall t \in \text{dom}(M) \text{\mid tidset}(\Upsilon_x), M_y(t) < M(t)) \]
   \[ \implies (W, S, \epsilon) \in F(S) \]
   By the transfinite induction principle, we only need to prove
   \[ \forall M, \xi_x, \xi_y, (\forall M', \xi_x', \xi_y', (\text{dom}(M') \text{\mid tidset}(\Upsilon_x'), M'_y) < (\text{dom}(M) \text{\mid tidset}(\Upsilon_x), M_y) \implies P(M', \xi_x', \xi_y', M_y)) \]
   \[ \implies P(M, \xi_x, \xi_y, M_y) \]
   The proof is by inversion over \( (M_y \models O^\omega_{\Upsilon_y}(W_y, S_y, \epsilon)) \). We only have two possible cases:
   i. \( (W_y, S_y) \xrightarrow{\tau_y} \text{skip} \) and \( \text{get\_obs}(\Upsilon_y) = \epsilon \):
      We get \( (W, S) \xrightarrow{\tau_x=\tau_y} \text{skip} \) and \( \text{get\_obs}(\Upsilon_x) = \epsilon \) by the definition of \( F \) (at the middle part in Figure 24), we know \( (W, S, \epsilon) \in F(S) \).
   ii. \( (W_y, S_y) \xrightarrow{\tau_x}^* (W'_y, S'_y), \text{get\_obs}(\Upsilon_x) = \epsilon, (M'_y \models O^\omega_{\Upsilon_y}(W'_y, S'_y, \epsilon)), \)
      \( \text{dom}(M_y) = \text{active\_Thrs}(W) \neq \emptyset \) and \( \forall t \in \text{dom}(M_y) \text{\mid tidset}(\Upsilon_x), M'_y(t) < M_y(t) \).
      We first show
      \[ (\text{dom}(M_y) \text{\mid tidset}(\Upsilon_x), M'_y) < (\text{dom}(M) \text{\mid tidset}(\Upsilon_x), M_y). \]
      (B.8)
      By the operational semantics, we know
      \( \text{dom}(M_y) = \text{active\_Thrs}(W) \subseteq \text{active\_Thrs}(W) = \text{dom}(M) \).
      Since \( \text{tidset}(\Upsilon_x) \subseteq \text{tidset}(\Upsilon_x \uplus \Upsilon'_x) \), we know \( \text{dom}(M_y) \text{\mid tidset}(\Upsilon_x \uplus \Upsilon'_x) \subseteq \text{dom}(M) \text{\mid tidset}(\Upsilon_x) \).
      If \( \text{dom}(M_y) \text{\mid tidset}(\Upsilon_x \uplus \Upsilon'_x) = \text{dom}(M) \text{\mid tidset}(\Upsilon_x) \), for any \( t \in \text{dom}(M_y) \text{\mid tidset}(\Upsilon_x \uplus \Upsilon'_x) \), we know \( t \in \text{dom}(M_y) \text{\mid tidset}(\Upsilon_x) \).
      Thus \( M'_y(t) < M_y(t) \).
      Thus we have proved (B.8).
      By the transfinite induction hypothesis, we have
4. \( \tau_{a} = \;: \tau_{s} \xrightarrow{\tau_{s}} + (W', S') \) \( \text{get \_obsb}(\tau_{s} :: \tau_{s}) = \epsilon \) \( (\forall \delta, \delta' \in \text{O}_M(W', S', \epsilon)) \& (\forall \delta \in \text{dom}(\delta) \setminus \text{tidset}(\tau_{s} :: \tau_{s})) \& (\forall \delta' \in \text{dom}(\delta) \setminus \text{tidset}(\tau_{s} :: \tau_{s})) \) \( \Rightarrow (W', S', \epsilon) \in F(S) \)

Thus we get \((W, S, \epsilon) \in F(S)\),

\[ \tau_{a} = \;: \tau_{s} \xrightarrow{\tau_{s}} + (W, S) \quad \text{get \_obsb}(\tau_{s} :: \tau_{s}) = \epsilon \quad \text{dom}(\delta) = \text{activeThrds}(W) \quad \Rightarrow \quad (W, S, \epsilon) \in F(S) \]

We want to prove \((W, S, \epsilon) \in F(S)\). We have two cases:

(a) activeThrds(W) \( \subseteq \text{tidset}(\tau_{s})\):

From \((\forall \delta, \delta' \in \text{O}_M(W, S), \epsilon)) \( \Rightarrow \) \((W, S, \epsilon) \in F(S)\), we know \((W, S, \epsilon) \in F(S)\). From the definition of \( F \) (at the middle part in Figure 24), we know \((W, S, \epsilon) \in F(S)\).

The order over the metrics, \( (\xi, \eta, \tau_{s}) \triangleleft (\xi', \eta', \tau'_{s}) \) if \((\xi, \eta, \tau_{s}) \triangleleft (\xi', \eta', \tau'_{s}) \) is defined as follows.

\[
(\xi, \eta, \tau_{s}) \triangleleft (\xi', \eta', \tau'_{s}) \quad \text{iff} \quad (\xi \subset \xi') \vee (\xi = \xi' \land \eta' = \emptyset) \land (\forall \tau \in \text{dom}(\tau), \tau_0(t) < M(t))
\]

It is clear that \((\xi, \eta, \tau_{s}) \triangleleft (\xi, \eta, \tau_{s}) \), \((\xi, \eta, \tau_{s}) \triangleleft (\xi, \eta, \tau_{s}) \), \((\xi, \eta, \tau_{s}) \triangleleft (\xi, \eta, \tau_{s}) \) is a well-founded order.

We view our goal as \((\forall M, \tau_{s}, \epsilon) \in M_M, \tau_{s}, \epsilon) \), which is reformulated as follows.

\[
(\forall M, \tau_{s}, \epsilon) \in M_M, \tau_{s}, \epsilon) \quad \text{dom}(\delta) = \text{activeThrds}(W) \quad \Rightarrow \quad (W, S, \epsilon) \in F(S)
\]

By the transfinite induction principle, we only need to prove

\[
(\forall M, \tau_{s}, \epsilon) \in M_M, \tau_{s}, \epsilon) \quad \text{dom}(\delta) = \text{activeThrds}(W) \quad \Rightarrow \quad P(M, \tau_{s}, \epsilon)
\]

The proof is by inversion over \((\forall M, \tau_{s}, \epsilon) \in M_M, \tau_{s}, \epsilon) \). We have four cases:

i. \((W, S, \epsilon) \in F(S)\), \( \tau_{s} : \tau_{s} \xrightarrow{\tau_{s}} \) \& \( \text{get \_obsb}(\tau_{s} :: \tau_{s}) = \emptyset \);

We get \((W, S) \in F(S)\).

ii. \((W, S, \epsilon) \in F(S)\), \( \tau_{s} : \tau_{s} \xrightarrow{\tau_{s}} \) \& \( \text{get \_obsb}(\tau_{s} :: \tau_{s}) = \emptyset \);

We get \((W, S) \in F(S)\).

iii. \((W, S, \epsilon) \in F(S)\), \( \tau_{s} : \tau_{s} \xrightarrow{\tau_{s}} \) \& \( \text{get \_obsb}(\tau_{s} :: \tau_{s}) = \emptyset \);

We get \((W, S) \in F(S)\).

With (B.8), by the transfinite induction hypothesis, we have

\[
(\forall W, S, \epsilon, \tau_{s}, \tau'_{s}, \tau'_s) \in (\forall W, S, \epsilon, \tau_{s}) \land (\forall W, S, \epsilon, \tau'_{s}) \land (\forall W, S, \epsilon, \tau'_s) \rightarrow (W, S, \epsilon) \in F(S)
\]

Thus we can get \((W, S, \epsilon) \in F(S)\).

iv. \((W, S, \epsilon) \in F(S)\), \( \tau_{s} : \tau_{s} \xrightarrow{\tau_{s}} \) \& \( \text{get \_obsb}(\tau_{s} :: \tau_{s}) = \emptyset \);

We get \((W, S) \in F(S)\).

With (B.8), by the transfinite induction hypothesis, we have

\[
(\forall W, S, \epsilon, \tau_{s}, \tau'_{s}) \in (\forall W, S, \epsilon, \tau_{s}) \land (\forall W, S, \epsilon, \tau'_{s}) \rightarrow (W, S, \epsilon) \in F(S)
\]

Thus we can get \((W, S, \epsilon) \in F(S)\).

Then we are done.
Proof of Lemma\[30\] First we define
\[
\text{roundsub}(\Xi, \xi, \Xi'_x) \iff (|\Xi| \neq \omega) \land (\Xi_x = \Xi) \lor \exists \Xi'. (\Xi = \Xi_x :: \Xi') \land (|\Xi| \neq \omega) \land (\xi \subseteq \text{tidset}(\Xi_x))
\]
\[
\text{minPos}(\Xi, \xi_1, \xi_2) \equiv \begin{cases} 
\text{minPos}(\Xi, \xi_2) & \text{if } \xi_1 = \emptyset \\
\text{minPos}(\Xi, \xi_1) & \text{if } \xi_1 \neq \emptyset
\end{cases}
\]
\[
\text{minPos}(\Xi, \xi) \equiv \begin{cases} 
1 & \text{if } \Xi = \epsilon \\
1 & \text{if } \Xi = \epsilon :: \Xi', |\Xi| \neq \omega \text{ and } \text{tid}(\xi) \in \xi \\
1 + \text{minPos}(\Xi', \xi) & \text{if } \Xi = \epsilon :: \Xi', |\Xi| \neq \omega \text{ and } \text{tid}(\xi) \notin \xi
\end{cases}
\]

We prove the following \([B.9]\) by inversion over \(\Xi \models O^\omega_{\rho}(W, S, \Xi_\rho)\).

If \(\Xi \models O^\omega_{\rho}(W, S, \Xi_\rho)\), then there exists \(\Xi_x\) such that \(\text{roundsub}(\Xi, \text{activeThrds}(W), \Xi_x)\).

(B.9)

Also, by inversion over \((W, S) \preceq (W, S) \circ (\mathcal{M}, \zeta)\), we know
\[
dom(\mathcal{M}) = \text{dom}(\zeta) = \text{activeThrds}(W) = \text{activeThrds}(W).
\]

Choose \(\Xi_x\) such that \(\text{roundsub}(\Xi, \text{activeThrds}(W), \Xi_x)\). Next we choose \(\mathcal{M}\) to be a function such that the following hold:

1. \(\dom(\mathcal{M}) = \text{activeThrds}(W)\).
2. For any \(t \in \dom(\mathcal{M})\), we have \(\mathcal{M}(t) = (\mathcal{M}(t), (\zeta(t), \text{dom}(\zeta) \setminus \{t\}), \text{minPos}(\Xi_x, \zeta(t), \{t\}))\).

We define the order \(\mathcal{M}'(t) < \mathcal{M}(t)\) as a dictionary order:
\[
\begin{align*}
(\mathcal{M}', (\xi', \xi_D), k') &< (\mathcal{M}, (\xi, \xi_D), k) \iff \\
(\mathcal{M}' < \mathcal{M}) &\lor (\mathcal{M}' = \mathcal{M} \land (\xi' \leq \xi) \land (k' = k)) \land (\mathcal{M}' = \mathcal{M} \land (\xi' < \xi) \land (\xi' \leq \xi_D) \land (k' = k)) \land (\mathcal{M}' = \mathcal{M} \land (\xi' = \xi) \land (\xi' \leq \xi_D) \land (\xi \leq \xi_D) \land (\xi' < \xi_D))
\end{align*}
\]

Clearly that \(\mathcal{M}'(t) < \mathcal{M}(t)\) is a well-founded order.

Next we prove: for any \(\Xi, W, S, \Xi_x, W, S, \mathcal{M}, \xi, \mathcal{M}\) and \(\Xi_x\), if

1. \(\Xi \models O^\omega_{\rho}(W, S, \Xi_\rho)\);
2. \((W, S) \preceq (W, S) \circ (\mathcal{M}, \zeta)\);
3. \(\text{roundsub}(\Xi, \text{activeThrds}(W), \Xi_x)\);
\[
\begin{align*}
\mathcal{M}(\mathcal{M}) = \text{activeThrds}(W) \quad &\text{and} \\
\text{for any } t \in \text{dom}(\mathcal{M}) &\text{, we have } \mathcal{M}(t) = (\mathcal{M}(t), (\zeta(t), \text{dom}(\zeta) \setminus \{t\}), \text{minPos}(\Xi_x, \zeta(t), \{t\})).
\end{align*}
\]

then \(\Xi \models (W, S) \preceq (W, S) \circ \mathcal{M}\).

By co-induction. We need to prove the following.

1. \((\mathcal{M}) = \text{activeThrds}(W) = \text{activeThrds}(W)\).
   \text{Proof:} From \((W, S) \preceq (W, S) \circ (\mathcal{M}, \zeta)\), we know \(\text{dom}(\mathcal{M}) = \text{dom}(\zeta) = \text{activeThrds}(W) = \text{activeThrds}(W)\).
2. \((W, S) \rightarrow (\mathcal{M}, \zeta)\) and \(\Xi = \varepsilon\), then
there exist \(\mathcal{T}\) and \(S'\) such that \(\text{get\_obs}(\mathcal{T}) = \varepsilon\) and \((W, S) \rightarrow^{T} + (\text{skip}, S')\).
   \text{Proof:} From \((W, S) \preceq (W, S) \circ (\mathcal{M}, \zeta)\), we know there exist \(\mathcal{T}\) and \(S'\) such that \(\text{get\_obs}(\mathcal{T}) = \varepsilon\) and \((W, S) \rightarrow^{T} + (\text{skip}, S')\).
3. If \((W, S) \rightarrow (\text{abort})\) and \(\Xi = \varepsilon\), then
there exist \(t\) and \(S\) such that \(t = (t, \text{clt. abort})\), \(t = \text{get\_obs}(\mathcal{T})\) and \((W, S) \rightarrow^{T} + \text{abort}\).
   \text{Proof:} From \((W, S) \preceq (W, S) \circ (\mathcal{M}, \zeta)\), we know there exist \(t\) and \(S\) such that \(t = (t, \text{clt. abort})\), \(t = \text{get\_obs}(\mathcal{T})\) and \((W, S) \rightarrow^{T} + \text{abort}\).
4. If \((W, S) \rightarrow (W', S')\) and \(\Xi = \varepsilon :: \mathcal{T}\), then there exist \(t, \mathcal{T}, W', S'\) and \(\mathcal{M}'\) such that all the following hold:
   \begin{enumerate}
   \item \((W, S) \rightarrow^{T} (W', S')\);
   \item \(t = \text{tid}(\xi)\), \(\text{get\_obs}(\xi) = \text{get\_obs}(\mathcal{T})\), \(\xi = (t, \text{term}) \Rightarrow (\xi = \text{last}(\mathcal{T}))\);
   \item \(\mathcal{T}' = (W', S') \preceq (W', S') \circ \mathcal{M}'\);
   \item for any \(t' \in \text{dom}(\mathcal{M})\), either \(t' \in \text{tidset}(\mathcal{T})\), or \(\mathcal{M}'(t') < \mathcal{M}'(t)\).
   \end{enumerate}
   \text{Proof:} From \((W, S) \preceq (W, S) \circ (\mathcal{M}, \zeta)\), we know there exist \(t, W, S', \mathcal{M}'\) and \(\zeta'\) such that all the following hold:
   \begin{enumerate}
   \item \((W, S) \rightarrow^{T} (W', S')\);
   \item \(t = \text{tid}(\xi)\), \(\text{get\_obs}(\xi) = \text{get\_obs}(\mathcal{T})\), \(\xi = (t, \text{term}) \Rightarrow (\xi = \text{last}(\mathcal{T}))\);
   \item \((W', S') \preceq (W', S') \circ (\mathcal{M}', \zeta')\);
   \end{enumerate}
Thus $M'(< t) < M(t)$. 

If $M'(t) = M(t)$ and $\emptyset \subseteq \zeta(t) \subseteq \zeta'(t)$, we prove $M'(t) < M(t)$ as follows. First we have $(\zeta'(t), \text{dom}(\zeta') \setminus \{ t \}) = (\zeta(t), \text{dom}(\zeta) \setminus \{ t \})$. 

Below we prove $\text{minPos}(I_x, \zeta, \zeta'(t), \{ t \}) = \text{minPos}(I_x, \zeta, \{ t \})$. We know 

$$\text{minPos}(I_x, \zeta, \zeta'(t), \{ t \}) = \text{minPos}(I_x, \zeta, \{ t \} \cup \text{tidset}(I_y)) = \text{minPos}(I_x, \zeta, \{ t \}) = \text{minPos}(I_x, \zeta, \{ t \})$$

Since roundsub$(\mathcal{S}, \text{activeThrds}(W'), I_x)$, we know there exists $I_y$ such that $I_y = I_x \cdot I_y$. Since $I_x \neq I_y$ and activeThrds$(W') \neq \emptyset$, we know $I_x \neq I_y$. Since $I_x \cdot I_y$, we know there exists $I_x$ such that $I_x = I_x \cdot I_y$ and $I_y = I_x \cdot I_y$. Since $I_x \neq \emptyset$ and $I_x = I_y$, we know $\text{minPos}(I_x, \zeta, \{ t \}) = \text{minPos}(I_x, \zeta, \{ t \})$. 

Thus $M'(t) < M(t)$. 

Next we prove: for any $t' \in \text{dom}(M) \setminus \{ t \}$, either $t' \in \text{tidset}(T)$, or $M'(t') < M(t')$. From (E), we know either $t' \in \text{tidset}(T)$, or $M'(t') < M(t')$. 

If $M'(t') < M(t')$, then $M'(t') < M(t')$. 

If $M'(t') = M(t')$ and $\emptyset \subseteq \zeta(t') \subseteq \zeta'(t')$, we prove $M'(t') < M(t')$ as follows. First we have $(\zeta'(t'), \text{dom}(\zeta') \setminus \{ t' \}) = (\zeta(t), \text{dom}(\zeta) \setminus \{ t' \})$. 

Below we prove $\text{minPos}(I_x, \zeta, \zeta'(t'), \{ t' \}) = \text{minPos}(I_x, \zeta, \{ t' \})$. We know 

$$\text{minPos}(I_x, \zeta, \zeta'(t'), \{ t' \}) = \text{minPos}(I_x, \zeta, \{ t' \}) = \text{minPos}(I_x, \zeta, \{ t' \}) = \text{minPos}(I_x, \zeta, \{ t' \})$$

Since roundsub$(\mathcal{S}, \text{activeThrds}(W'), I_x)$, we know there exists $I_y$ such that $I_y = I_x \cdot I_y$. Since $I_x \neq I_y$ and activeThrds$(W') \neq \emptyset$, we know $I_x \neq I_y$. Since $I_x \cdot I_y$, we know there exists $I_x$ such that $I_x = I_x \cdot I_y$ and $I_y = I_x \cdot I_y$. Since $t \neq t'$ and $t = \text{tid}([t])$, we know 

$$\text{minPos}(I_x, \zeta', \{ t \}) = \text{minPos}(I_x, \zeta', \{ t \})$$

Next we prove roundsub$(\mathcal{S}', \{ t' \}, I_y)$. From roundsub$(\mathcal{S}, \text{activeThrds}(W'), I_x)$, we have two cases: 

- If $I_x \neq I_y$ and $I_y = I_x$, then $I_y = I_x$. 
- If $I_x \neq I_y$ and activeThrds$(W') \subseteq \text{tidset}(I_x)$, then $I_y \neq I_x$ and $\{ t' \} \subseteq \text{tidset}(I_y)$. Since $I_x = I_x \cdot I_y$ and $t = \text{tid}([t])$, we know tidset$(I_x) = \{ t \} \cup \text{tidset}(I_y)$. Since $t \neq t'$, we know $\{ t \} \not\subseteq \text{tidset}(I_y)$.
Thus we are done.  

Lemma 34. If $\xi \neq \emptyset$, roundsub$(\bar{T}, \xi', \bar{T}' \subseteq \text{activeThrds}(W'))$, then roundsub$(\bar{T}, \xi', \bar{T}' \subseteq \text{activeThrds}(W'))$, we know roundsub$(\bar{T}, \xi, \bar{T} \subseteq \text{activeThrds}(W))$, then by Lemma 34 we know $\minPos(\bar{T}, \xi, \{t\}) = \minPos(\bar{T}', \xi, \{t\})$.

Thus $M'(t) < M(t)$.

- If $M'(t) = \bar{M}(t), \emptyset < \xi' \subseteq \xi'$ and $t \notin \xi'$, we prove $M'(t') < M(t')$ as follows. First we have $(\xi', \xi') = (\xi' \subseteq \xi')$.

Below we prove $\minPos(\bar{T}, \xi, \xi', \{t\}) < \minPos(\bar{T}', \xi, \xi', \{t\})$. We know $\minPos(\bar{T}, \xi, \xi', \{t\}) = \minPos(\bar{T}', \xi, \xi', \{t\}) = \minPos(\bar{T}, \xi, \{t\})$.

Since roundsub$(\bar{T}, \xi, \text{activeThrds}(W), \bar{T} \subseteq \text{activeThrds}(W'))$, we know there exists $\bar{T} \subseteq \bar{T}'$ such that $\bar{T} = \bar{T}' \subseteq \bar{T}_x$.

Thus we prove roundsub$(\bar{T}, \xi, \xi', \bar{T}_x)$. From roundsub$(\bar{T}, \xi, \text{activeThrds}(W), \bar{T} \subseteq \text{activeThrds}(W'))$, we have two cases:

- If $|\bar{T}| \neq \omega$ and $\bar{T}_x = \emptyset$, then $\bar{T}_x = \emptyset$. Since $\bar{T}_x = \emptyset$, we know $\bar{T}_x = \emptyset$. Then $t = \text{tid}(\bar{T}_x)$.

Thus roundsub$(\bar{T}', \xi', \{t\})$. On the other hand, since roundsub$(\bar{T}', \xi, \text{activeThrds}(W'), \bar{T}_x') \subseteq \text{activeThrds}(W')$, we know roundsub$(\bar{T}', \xi, \xi', \bar{T}_x')$. Then by Lemma 34 we know $\minPos(\bar{T}, \xi, \xi', \{t\}) = \minPos(\bar{T}', \xi, \xi', \{t\})$.

Thus $M'(t) < M(t)$.

Thus we are done. □

Lemma 32. If $(\pi, \bar{T}_x) \models C_x(W, S, \bar{T}_x)$, then there exists $\pi_k$ such that $\pi_k = \rho \cdot \rho_{\text{obs}(i)} :: \pi_k$.

Lemma 33. If $(\pi, \bar{T}_x) \models C_x(W, S, \bar{T}_x)$, $(W, S) \models \rho_{\text{obs}(i)} (W, S)$ and $\pi_k = \rho \cdot \rho_{\text{obs}(i)} :: \pi_k$, then $\pi_k \models C_x(W, S, \bar{T}_x)$.

Proof. By co-induction, and then by inversion three times over (\::\bar{T}_x) \models C_x(W, S, \bar{T}_x).

Lemma 34. If $\xi \neq \emptyset$, roundsub$(\bar{T}, \xi, \bar{T}_x)$ and roundsub$(\bar{T}, \xi, \bar{T}_x)$, then $\minPos(\bar{T}, \xi, \xi) = \minPos(\bar{T}, \xi, \xi)$. □

Proof. From the premises, we know $|\bar{T}_x| \neq \omega$ and $|\bar{T}_y| \neq \omega$. Suppose $|\bar{T}_x| \leq |\bar{T}_y|$. We have two cases:

- $|\bar{T}| \neq \omega$ and $\bar{T}_x = \bar{T}$:

We know there exists $\bar{T}_y$ such that $\bar{T} = \bar{T}_y :: \bar{T}_y$. Thus $\bar{T}_y = \emptyset$. Then we have $\bar{T}_x = \bar{T}_y$ and we are done.

- there exists $\bar{T}_x'$ such that $\bar{T} = \bar{T}_x :: \bar{T}_x'$ and $\xi' \subseteq \text{tidset}(\bar{T}_x)$:

We know there exists $\bar{T}_y'$ such that $\bar{T} = \bar{T}_y' :: \bar{T}_y'$. Thus there exists $\bar{T}_x$ such that $\bar{T}_y = \bar{T}_x :: \bar{T}_y'$ and $\bar{T}_x = \bar{T}_x :: \bar{T}_x'$. Since $\xi \neq \emptyset$, we know $\text{tidset}(\bar{T}_x) \neq \emptyset$. Thus there exist $e$ and $\bar{T}_0$ such that $\bar{T}_x = e :: \bar{T}_0$. Thus $\bar{T}_y = e :: \bar{T}_0$. By induction over $|\bar{T}_0|$. □
Figure 24 defines F and $\text{gfp}\ F$ (i.e., $\mathcal{O}_{\text{tr}}^\infty$ at the bottom part of the figure).

\[
S \overset{\text{def}}{=} \{(M, W, S, \mathcal{I}_0) \mid \exists \mathcal{I}, W, S. ((\mathcal{I} \models \mathcal{O}_{\text{tr}}(W, S, \mathcal{I}_0)) \wedge (\mathcal{I} \models (W, S) \leq (W, S) \circ M))\}.
\]

So from the co-induction principle, we only need to prove:

\[
S \subseteq F(S), \ i.e. \ \forall M, W, S, \mathcal{I}_0. (M, W, S, \mathcal{I}_0) \in S \implies (M, W, S, \mathcal{I}_0) \in F(S).
\]

By unfolding S and by inversion over $\mathcal{O}_{\text{tr}}^\infty$, we have the following cases.

1. $(\mathcal{I} \models (W, S) \leq (W, S) \circ M), n > 0, (W, S) \xrightarrow{\mathcal{I}}^n \text{abort}$ and get_obs($\mathcal{I}$) = $\mathcal{I}_0$:

2. $(\mathcal{I} \models (W, S) \leq (W, S) \circ M), n > 0, (W, S) \xrightarrow{\mathcal{I}}^n \text{skip}_{-\_}$ and get_obs($\mathcal{I}$) = $\mathcal{I}_0$:

3. $(\mathcal{I} \models (W, S) \leq (W, S) \circ M), n > 0, \mathcal{I} = \mathcal{I}_x :: \mathcal{I}_y, \mathcal{I}_0 = \epsilon, (W, S) \xrightarrow{\mathcal{I}}^n (W_y, S_y),$

   get_obs($\mathcal{I}_x$) = $\epsilon$, (\mathcal{I}_y \models \mathcal{O}_{\text{tr}}((W_y, S_y, \epsilon))$, activeThrds($W$) \subseteq \text{tidset}($\mathcal{I}_x$)):

To prove $(M, \mathcal{W}, S, \mathcal{I}_0) \in F(S)$, we want to prove: there exist $T_x, W_y, S_y$ and $M_y$ such that

\[
(W, S) \xrightarrow{\mathcal{I}}^* (W_y, S_y), \text{ get_obs}(\mathcal{I}_x) = \epsilon, (M_y, W_y, S_y, \epsilon) \in S, \text{ dom}(M) = \text{activeThrds}(W) \neq \emptyset, \forall t \in \text{dom}(M) \setminus \text{tidset}(\mathcal{I}_x), M_y(t) < M(t) .
\]

Since $(\mathcal{I}_y \models \mathcal{O}_{\text{tr}}((W_y, S_y, \epsilon))$, we only need to prove:

\[
\forall n, \mathcal{I}, W, S, M, \mathcal{I}_x, \mathcal{I}_y, W_y, S_y, \mathcal{I}_0.
\]

\[
(n > 0) \wedge (\mathcal{I} \models (W, S) \leq (W, S) \circ M) \wedge (\mathcal{I} = \mathcal{I}_x :: \mathcal{I}_y) \wedge ((W, S) \xrightarrow{\mathcal{I}}^n (W_y, S_y)) \wedge (\text{get_obs}(\mathcal{I}_x) = \mathcal{I}_0) \implies \exists T_x, W_y, S_y, M_y .
\]

\[
(W, S) \xrightarrow{\mathcal{I}}^* (W_y, S_y)) \wedge (\text{get_obs}(\mathcal{I}_x) = \mathcal{I}_0) \wedge (\text{dom}(M) = \text{activeThrds}(W) \neq \emptyset) \wedge (\forall t \in \text{dom}(M) \setminus \text{tidset}(\mathcal{I}_x), M_y(t) < M(t) ) .
\]

By induction over $n$.

- **Base case**: $n = 1$. Suppose $\mathcal{I}_x = \epsilon :: \epsilon$. Since $\mathcal{I} \models (W, S) \leq (W, S) \circ M$, we know dom($M$) = activeThrds($W$) = activeThrds($W$) $\neq \emptyset$. Also there exist $t, \mathcal{I}, \mathcal{W}', \mathcal{S}'$ and $M'$ such that the following hold:

   \[
   (A) \ (W, S) \xrightarrow{\mathcal{I}}^* (W', S');
   \]

   \[
   (B) t = \text{tid}(\mathcal{e}), \text{get_obs}(\mathcal{e}) = \text{get_obs}(\mathcal{T}), (\mathcal{e} = (t, \mathcal{\text{term}})) \Rightarrow (\mathcal{e} = \text{last}(\mathcal{T}));
   \]

   \[
   (C) \mathcal{I}_y \models (W_y, S_y) \leq (W', S') \circ M';
   \]

   \[
   (D) \text{for any } t' \in \text{dom}(M), \text{either } t' \in \text{tidset}(\mathcal{T}), \text{or } M'(t') < M(t').
   \]

From (D), we have: $\forall t \in \text{dom}(M) \setminus \text{tidset}(\mathcal{T}), M'(t) < M(t)$.

- **Inductive step**: $n = k + 1$ and $k > 0$. Suppose $\mathcal{I}_x = \epsilon :: \mathcal{I}_x'$ and

   \[
   (W, S) \xrightarrow{\mathcal{I}}^* (W_x, S_x) \wedge (W_x, S_x) \xrightarrow{\mathcal{I}}^{k+1} (W_y, S_y) .
   \]

Since $\mathcal{I} \models (W, S) \leq (W, S) \circ M$, we know dom($M$) = activeThrds($W$) = activeThrds($W$) $\neq \emptyset$. Also there exist $t, \mathcal{I}, \mathcal{W}', \mathcal{S}'$ and $M'$ such that all the following hold:

   \[
   (A) \ (W, S) \xrightarrow{\mathcal{I}}^* (W', S');
   \]

   \[
   (B) t = \text{tid}(\mathcal{e}), \text{get_obs}(\mathcal{e}) = \text{get_obs}(\mathcal{T}), (\mathcal{e} = (t, \mathcal{\text{term}})) \Rightarrow (\mathcal{e} = \text{last}(\mathcal{T}));
   \]

   \[
   (C) \mathcal{I}_y \models (W_y, S_y) \leq (W', S') \circ M';
   \]

   \[
   (D) \text{for any } t' \in \text{dom}(M), \text{either } t' \in \text{tidset}(\mathcal{T}), \text{or } M'(t') < M(t').
   \]

From (D), we have: $\forall t \in \text{dom}(M) \setminus \text{tidset}(\mathcal{T}), M'(t) < M(t)$.

From the induction hypothesis, we have:

\[
\exists T_x, W_y, S_y, M_y,
\]

\[
((W_x', S_x') \xrightarrow{\mathcal{I}_x'} (W_y, S_y)) \wedge (\text{get_obs}(\mathcal{T}_x') = \text{get_obs}(\mathcal{T}_x')) \wedge (\mathcal{I}_x \models (W_y, S_y) \leq (W_y, S_y) \circ M_y) \wedge (\text{dom}(M') = \text{activeThrds}(W') \neq \emptyset) \wedge (\forall t \in \text{dom}(M') \setminus \text{tidset}(\mathcal{T}_x), M_y(t) < M(t))
\]

Thus we have $\mathcal{I} \models (W, S) \leq (W, S) \circ M$ and get_obs($\mathcal{T} :: \mathcal{I}_x$) = get_obs($\mathcal{e} :: \mathcal{I}_e$) = get_obs($\mathcal{I}_x$). Also, for any $t \in \text{dom}(M) \setminus \text{tidset}(\mathcal{T} :: \mathcal{I}_x)$, we know $t \in \text{dom}(M') \setminus \text{tidset}(\mathcal{T}_x)$. Thus $M'(t) < M(t)$. Also, we know $t \in \text{dom}(M') \setminus \text{tidset}(\mathcal{T}_x)$.

Thus $M_y(t) < M(t)$.

4. $(\mathcal{I} \models (W, S) \leq (W, S) \circ M), n > 0, \mathcal{I} = \mathcal{I}_x :: \mathcal{I}_y, \mathcal{I}_0 = \epsilon :: \mathcal{I}_b, (W, S) \xrightarrow{\mathcal{I}}^n (W_y, S_y),$

   get_obs($\mathcal{I}_x$) = $\epsilon :: \mathcal{I}_a, \mathcal{I}_y \models \mathcal{O}_{\text{tr}}((W_y, S_y, \epsilon))$, activeThrds($W$) $\subseteq \text{tidset}($\mathcal{I}_x$):

To prove $(M, \mathcal{W}, S, \mathcal{I}_0) \in F(S)$, we want to prove: there exist $T_x, W_y, S_y$ and $M_y$ such that
\[(\mathcal{W}, \mathcal{S}) \xrightarrow{\tau_a} (\mathcal{W}_y, \mathcal{S}_y), \text{ get\_obs}v(T_x) = e_a :: \Xi_a, (\mathcal{M}_y, \mathcal{W}_y, \mathcal{S}_y, \Xi_b) \in S,\]
\[\text{dom}(\mathcal{M}) = \text{activeThrs}(\mathcal{W}) \neq \emptyset, \forall t \in \text{dom}(\mathcal{M}) \setminus \text{tidset}(T_x). \mathcal{M}_y(t) < \mathcal{M}(t).\]

Since \((\mathcal{S}_y \models \mathcal{O}^{\text{co}}_{\mathcal{W}}(\mathcal{W}_y, \mathcal{S}_y, \Xi_b))\), we are done by (B.10).
B.5 Equivalence between Contextual Refinement and Starvation-Freedom/Deadlock-Freedom

Below we first define starvation-freedom. Then we define linearizability using histories, which are finite event traces only. We prove the equivalence result (Theorem 42) is the basis of our new results that relate \( \Pi \subseteq \rho \Gamma \) to starvation-freedom and deadlock-freedom of linearizable objects.

Finally, we show the new equivalence results (Theorems 43 and 44) between linearizability, starvation-freedom, deadlock-freedom and the contextual refinement under fair scheduling.

Formalizing starvation-freedom and deadlock-freedom. Following the earlier work \[8\][21][22], we define starvation-freedom and deadlock-freedom over execution traces. Then we introduce some auxiliary definitions in Fig. 25. We use \( \mathcal{T}_o \) \( [W,S] \) to represent the set of whole event traces generated by executions starting from \( ([W],S) \). Here \( ([W],S) \xrightarrow{\omega} ([W'],S') \) represents a zero or multi-step execution of \( ([W],S) \) that leads to \( ([W'],S') \) with the sequence \( T \) of events generated; \( ([W],S) \xrightarrow{\omega} \) then represents an infinite execution with the whole event trace \( T \), which must be infinite too. We also insert a \( (\text{spawn},n) \) event at the head of each event trace to record the number of threads in the program. The lifting \( [W] \) appends an \( \text{end} \) command at the end of each thread, which generates a \( (t,\text{term}) \) event to mark the termination of this thread \( t \).

Besides, we say \( T \) is fair, i.e., \( \text{fair}(T) \), if \( T \) is finite or every non-terminating thread \( t \) has infinite steps on the trace. Here \( T|t \) is the subsequence of \( T \) consisting of events from thread \( t \) only.

Also in Fig. 25 we define \( \text{prog-t}(T) \) to say that every method call in \( T \) eventually finishes. Here \( \text{pend_inv}(T(1..i)) \) gets the set of pending invocation events in the sub-trace \( T(1), \ldots, T(i) \) of \( T \), and \( \text{match}(e,T(j)) \) says that \( T(j) \) is a return event which matches with the invocation \( e \), so the corresponding method call finishes. \( \text{prog-s}(T) \) requires that some pending method call finishes. Different from \( \text{prog-t}(T) \), the return event \( T(j) \) in \( \text{prog-s}(T) \) does not have to be a matching return of the pending invocation \( e \). We also write \( \text{abt}(T) \) to say that \( T \) ends with a fault event.

Starvation-freedom and deadlock-freedom require \( \text{prog-t} \) and \( \text{prog-s} \), respectively, in every fair execution. They are defined below.

Definition 35 (Starvation-free objects). \( \text{starvation-free}_{\rho}(\Pi) \) iff

\[
\forall n, C_1, \ldots, C_n, \sigma, \sigma_0, T. T \in \mathcal{T}_o[[\text{let } \Pi \in C_1 \ldots \ Parallel C_n], (\sigma, \sigma_0, \ominus)] \wedge (\sigma_0, \ominus) \models P \wedge \text{fair}(T) \\
\implies \text{prog-t}(T) \lor \text{abt}(T).
\]

Here \( \ominus = \{ t_1 \sim o, \ldots, t_n \sim o \} \).

Definition 36 (Deadlock-free objects). \( \text{deadlock-free}_{\rho}(\Pi) \) iff

\[
\forall n, C_1, \ldots, C_n, \sigma, \sigma_0, T. T \in \mathcal{T}_o[[\text{let } \Pi \in C_1 \ldots \ Parallel C_n], (\sigma, \sigma_0, \ominus)] \wedge (\sigma_0, \ominus) \models P \wedge \text{fair}(T) \\
\implies \text{prog-s}(T) \lor \text{abt}(T).
\]

Formalizing linearizability. Linearizability describes atomic behaviors of object implementations. Following its standard definition \[15\], we define linearizability using histories, which are finite event traces \( T \) consisting of only method invocations, returns, and object faults. As defined in Fig. 25, we say a return \( e \) matches an invocation \( e_1 \), denoted as \( \text{match}(e_1,e_2) \), if they have the same thread ID. An invocation is \( \text{pending} \) in \( T \) if no matching return follows it. We use \( \text{pend_inv}(T) \) to get the set of pending invocations in \( T \). We complete a history \( T \) by appending zero or more return events, and dropping the remaining pending invocations. The result is denoted by \( \text{completions}(T) \). It is a set of histories, and for each history in it, every invocation has a matching return event.

Definition 37 (Linearizable histories). \( T \xrightarrow{\text{lin}} T' \) iff both the following hold.

1. \( \forall t. T|t = T'|t \).
2. There exists a bijection \( \pi : \{1, \ldots, |T|\} \rightarrow \{1, \ldots, |T'|\} \) such that \( \forall i. T(i) = T'(\pi(i)) \) and

\[
\forall i,j. i < j \land \text{is_res}(T(i)) \land \text{is_inv}(T(j)) \implies \pi(i) < \pi(j).
\]

That is, \( T \) is linearizable with respect to \( T' \) if the latter is a permutation of the former, preserving the order of events in the same threads and the order of the non-overlapping method calls. Then an \( \text{object} \) is linearizable iff each of its concurrent histories after \( \text{completions} \) is linearizable with respect to some \( \text{legal sequential history} \). As defined in Fig. 25 we use \( \Gamma \triangleright (\Sigma,T') \) to mean that \( T' \) is a legal sequential history generated by any client using the specification \( \Gamma \) with an abstract initial state \( \Sigma \).

Definition 38 (Extensions of a history). \( \text{extensions}(T) \) is a set of well-formed histories where we extend \( T \) by appending return events. It is inductively defined as follows.

\[
\text{well-formed}(T) \\
T \in \text{extensions}(T) \\
T' \in \text{extensions}(T) \quad \text{is_ret}(e) \quad \text{well-formed}(T::e) \\
T'::e \in \text{extensions}(T)
\]

Definition 39 (Completions of a history). \( \text{truncqe}(T) \) is the maximal complete sub-history of \( T \). It is inductively defined by dropping the pending invocations in \( T \).

\[
\text{truncqe}(e) \overset{\text{def}}{=} e \\
\text{truncqe}(e::T) \overset{\text{def}}{=} \{
\begin{array}{ll}
e::\text{truncqe}(T) & \text{if is_res}(e) \text{ or } \exists t. \text{match}(e,T(t)) \\
\text{truncqe}(T) & \text{otherwise}
\end{array}
\}
\]

Then \( \text{completions}(T) \overset{\text{def}}{=} \{ \text{truncqe}(T') \mid T' \in \text{extensions}(T) \} \).
\( \mathcal{T}_o[W,S] \overset{\text{def}}{=} \{ (\text{spawn},[W])::T \mid ([W],S)\xrightarrow{T} \omega \}
\)

\( \lor ([W],S)\xrightarrow{T}\ast (\text{skip},\_)\lor ([W],S)\xrightarrow{T}\ast \text{abort} \)

\[ \text{let } \Pi \text{ in } C_1\ldots\parallel C_n \overset{\text{def}}{=} \text{let } \Pi \text{ in } (C_1;\text{end})\parallel\ldots\parallel(C_n;\text{end}) \]

\[ \text{let } \Pi \text{ in } C_1\parallel\ldots\parallel C_n \overset{\text{def}}{=} n \]

> \text{tnum}((\text{spawn},n)::T) \overset{\text{def}}{=} n

\( \text{fair}(T) \text{ if } |T| = \omega \implies \forall t \in [1..\text{tnum}(T)].[|T_t|] = \omega \lor \text{last}(T_t) = (t, \text{term}) \)

\( \text{match}(e_1,e_2) \overset{\text{def}}{=} \text{is_inv}(e_1) \land \text{is_res}(e_2) \land \text{tid}(e_1) = \text{tid}(e_2) \)

\[ \text{pend_inv}(T) \overset{\text{def}}{=} \{ e \mid \exists i. e = T(i) \land \text{is_inv}(e) \land \neg \exists j. (j > i \land \text{match}(e,T(j))) \} \]

\( \text{prog-t}(T) \text{ if } \forall i, e \in \text{pend_inv}(T(1..i)) \implies \exists j > i \land \text{match}(e,T(j)) \)

\( \text{prog-s}(T) \text{ if } \forall i, e \in \text{pend_inv}(T(1..i)) \implies \exists j > i \land \text{is_res}(T(j)) \)

\( \text{abt}(T) \text{ if } \exists i. \text{is_abt}(T(i)) \)

\[ \mathcal{O}[[W,S]] \overset{\text{def}}{=} \{ \text{get_obs}(T) \mid T \in \mathcal{T}_o[W,S] \land \text{fair}(T) \} \]

\[ \mathcal{O}([[W,S]] \overset{\text{def}}{=} \{ \text{get_obs}(T) \mid T \in \mathcal{T}_o[W,S] \} \]

\[ \mathcal{T}[[W,S]] \overset{\text{def}}{=} \{ T \mid \exists W',s'. (W,S)\xrightarrow{T}\ast (W',s') \lor (W,S)\xrightarrow{T}\ast \text{abort} \} \]

\[ \mathcal{H}[[W,S]] \overset{\text{def}}{=} \{ \text{get_hist}(T) \mid T \in \mathcal{T}[[W,S]] \} \]

\[ \mathcal{O}[[W,S]] \overset{\text{def}}{=} \{ \text{get_obs}(T) \mid T \in \mathcal{T}[[W,S]] \} \]

\[ \mathcal{T}_o,\mathcal{T}',\mathcal{T}_{\text{e}} \in \text{completions}(T) \land \Gamma \overset{\text{def}}{=} (\Sigma,T') \land \mathcal{T}_{\text{e}}\leqslant_{\text{lin}} T' \]

\[ \Gamma \overset{\text{def}}{=} (\Sigma,T) \text{ if } \exists n, C_1,\ldots,C_n, \sigma_c, \sigma, \Sigma. T \in \mathcal{H}[[\text{let } \Pi \text{ in } C_1\parallel\ldots\parallel C_n,(\sigma_c,\Sigma,\odot)]] \land \text{seq}(T) \]

**Figure 25.** Event traces.

**Definition 40 (Linearizability of objects).** The object implementation \( \Pi \) is linearizable with respect to \( \Gamma \), written as \( \Pi \leqslant_{\text{lin}} \Gamma \), if

\[ \forall n, C_1,\ldots,C_n, \sigma_c, \sigma, \Sigma. T \in \mathcal{H}[[\text{let } \Pi \text{ in } C_1\parallel\ldots\parallel C_n,(\sigma_c,\Sigma,\odot)]] \land ((\sigma,\Sigma) \models P) \]

\[ \implies \exists T_e, T_e \in \text{completions}(T) \land \Gamma \overset{\text{def}}{=} (\Sigma,T_e) \land T_e \leqslant_{\text{lin}} T' \]

\[ \text{Equivalence results.} \]

**Definition 41 (Basic contextual refinement).** \( \Pi \leqslant_{\text{co}} \Pi' \) if

\[ \forall n, C_1,\ldots,C_n, \sigma_c, \sigma, \Sigma. ((\sigma,\Sigma) \models P) \implies \mathcal{O}[[\text{let } \Pi \text{ in } C_1\parallel\ldots\parallel C_n,(\sigma_c,\sigma,\odot)] \subseteq \mathcal{O}[[\text{let } \Pi' \text{ in } C_1\parallel\ldots\parallel C_n,(\sigma_c,\sigma,\odot)]] \].

**Theorem 42 (Basic equivalence for linearizability).** \( \Pi \leqslant_{\text{lin}} \Gamma \implies \Pi \leqslant_{\text{co}} \Gamma \).

**Theorem 43 (1 in Fig. 18).** \( \Pi \leqslant_{\text{co}} \Gamma \land \text{starvation-free}_{P}(\Pi) \implies \Pi \leqslant_{\text{co}} \Gamma \).

**Theorem 44 (2 in Fig. 18).** \( \Pi \leqslant_{\text{lin}} \Gamma \land \text{deadlock-free}_{P}(\Pi) \iff \Pi \leqslant_{\text{wr}_1(\Pi)} \text{wr}_1(\Gamma) \).

In the following subsections, we give the proofs of the above equivalence theorems.

#### B.5.1 Most general client

The key in our proofs is the use of the Most General Client (MGC). Informally, an MGC is a special client which itself can produce all the possible behaviors produced by any clients. We can define the MGC versions of linearizability, progress properties, and observable refinements, and prove their relationships to the original definitions, which universally quantify over arbitrary client programs. Then we reduce the problems of proving the equivalence between original definitions to proving some relationships between the MGC versions. Since an MGC is a specific client, the latter task is usually much simpler.

In fact, we define three MGCs, which produce different “general” behaviors. We assume \( \text{dom}(\Pi) = \{ f_1,\ldots,f_m \} \), and introduce two instructions to get a random (non-deterministic) value. \( x := \text{rand}(m) \) assigns \( x \) a random integer \( i \in [1..m] \), and \( x := \text{rand}(\) computes an arbitrarily large random integer. Then, for any \( n \), we define \( \text{MGC}_n \) as follows.

\[ \text{MGT}_t \overset{\text{def}}{=} \text{while } (\text{true}) \{
\]

\[ x_t := \text{rand}(); \ y_t := \text{rand}(m); \ z_t := f_{y_t}(x_t); \}

\[ \text{MGC}_n \overset{\text{def}}{=} \bigwedge_{i \in [1..n]} \text{MGT}_t \]
Here \(x_t, y_t\) and \(z_t\) are all local variables for thread \(t\). Each thread in MGC\(_n\) keeps calling a random method with a random argument. We also define MGC\(_p_n\), which print out the arguments and return values for method calls.

\[
\text{MGT}_p_t \overset{\text{def}}{=} \text{while (true)\{}
x_t := \text{rand}();
y_t := \text{rand}(m);
\text{print}(y_t, x_t);
z_t := f_{y_t}(x_t);
\text{print}(z_t);
\text{\} }
\]

\[
\text{MGC}_p_n \overset{\text{def}}{=} \|_{\ell \in [1..n]} \text{MGT}_p_t
\]

The third MGC MGC\(_p_1\) is useful to observe the progress of objects. Each thread non-deterministically decides whether to continue calling the methods or to end itself. Also it always prints out 0 before a method call and prints out 1 when the method call finishes.

\[
\text{MGT}_p_1 \overset{\text{def}}{=} \text{while (\text{rand}() > 0)\{}
x_1 := \text{rand}();
y_1 := \text{rand}(m);
\text{print}(0);
z_1 := f_{y_1}(x_1);
\text{print}(1);
\text{\} }
\]

\[
\text{MGC}_p_1 \overset{\text{def}}{=} \|_{\ell \in [1..n]} \text{MGT}_p_1
\]

The client memory for the above MGCs should contain the local variables for each thread.

\[
\sigma_{\text{MGC}(n)} \overset{\text{def}}{=} \{x_t \sim \_ , y_t \sim \_ , z_t \sim \_ | 1 \leq t \leq n\}
\]

### B.5.2 Basic equivalence for linearizability

Follow earlier work \cite{22}, we first define the MGC versions of “linearizability” and “refinement.”

\textbf{Definition 45.} \(\Pi \preceq^M \text{MGC} \Gamma\) iff

\[
\forall n, \sigma_{\text{MGC}(n)}, \sigma, \Sigma, T, T' \in \mathcal{H}[\{\\text{let } \Pi \text{ in MGC}_n\}, \sigma_{\text{MGC}(n)}, \sigma, \odot]] \land (\sigma, \Sigma) \models P \implies \exists T_c, T'_c \in \text{completions}(T) \land \Gamma \triangleright_n (\sigma_{\text{MGC}(n)}, \Sigma, T') \land T_c \preceq^\text{lin} T'_c.
\]

where

\[
\Gamma \triangleright_n (\sigma_{\text{MGC}(n)}, \Sigma, T) \overset{\text{def}}{=} T \in \mathcal{H}[\{\\text{let } \Gamma \text{ in MGC}_n\}, \sigma_{\text{MGC}(n)}, \Sigma, \odot]] \land \text{seq}(T).
\]

\textbf{Definition 46.} \(\Pi \preceq^P \Gamma\) iff

\[
\forall n, \sigma_{\text{MGC}(n)}, \sigma, \Sigma, ((\sigma, \Sigma) \models P) \implies \mathcal{H}[\{\\text{let } \Pi \text{ in MGC}_n\}, \sigma_{\text{MGC}(n)}, \sigma, \odot]] \subseteq \mathcal{H}[\{\\text{let } \Gamma \text{ in MGC}_n\}, \sigma_{\text{MGC}(n)}, \Sigma, \odot]]
\]

To prove Theorem \[42\] we prove the following lemmas.

\textbf{Lemma 47.} \(\Pi \preceq^P \Gamma \iff \Pi \preceq^M \text{MGC} \Gamma\).

\textbf{Lemma 48.} \(\Pi \preceq^M \text{MGC} \Gamma \iff \Pi \preceq^P \Gamma\).

\textbf{Lemma 49.} \(\Pi \preceq^P \Gamma \iff \Pi \preceq^M \text{MGC} \Gamma\).

\textbf{Proof of Lemma 47.} We can see \(\Pi \preceq^M \text{MGC} \Gamma\) is simply defined by fixing the arbitrarily quantified client programs in \(\Pi \preceq^P \Gamma\) (Definition \(40\)) as MGC\(_n\). Intuitively, the key to prove this lemma is to show that any history generated by an arbitrary client program can also be generated by MGC, as shown in the following lemma.

\textbf{Lemma 50 (MGC is the most general).} For any \(n, C_1, \ldots, C_n, \sigma_c, \sigma_{\text{MGC}(n)}\), and \(\sigma_o\), we have

\[
\mathcal{H}[\{\\text{let } \Pi \text{ in } C_1 \parallel \ldots \parallel C_n\}, \sigma_c, \sigma_o, \odot]] \subseteq \mathcal{H}[\{\\text{let } \Pi \text{ in MGC}_n\}, \sigma_{\text{MGC}(n)}, \sigma_o, \odot]]
\]

\textbf{Proof.} We construct a simulation \(\preceq^M\) between the client program and the MGC. We need the simulation relation to satisfy the following (B.11).

For any \(W_1, S_1, W_2, S_2\) and \(e_1\), if \((W_1, S_1) \preceq^M (W_2, S_2)\), then

1. if \((W_1, S_1) \overset{e_1}{\rightarrow} \text{abort}\) and is_obj_abt\(\left(e_1\right)\), then there exists \(T_2\) such that \((W_2, S_2) \overset{T_2}{\rightarrow} \text{abort}\) and \(e_1 = \text{get_hist}(T_2)\);
2. if \((W_1, S_1) \overset{e_1}{\rightarrow} \left(W'_1, S'_1\right)\), then there exist \(T_2, W'_2\) and \(S'_2\) such that \((W_2, S_2) \overset{T_2}{\rightarrow} \left(W'_2, S'_2\right)\), get_hist\(\left(e_1\right) = \text{get_hist}(T_2)\) and \((W'_1, S'_1) \preceq^M (W'_2, S'_2)\).

(B.11)

The simulation relation is constructed as shown in Fig. \[25\]. That is, for each client thread \(t\), if the left side is in some normal client code, it corresponds to MGT\(_t\) at the right side; otherwise, if the left side is inside a method call, its code is the same as the right side. Informally, the following hold for each thread \(t\).

1. Each normal client step of the left corresponds to zero step of MGT\(_t\).
2. Each method invocation corresponds to the steps executing MGT\(_t\) to the same method body, with the same argument.
(3) Each step inside the method body at the left corresponds to the same step at the right.
(4) Each return step at the left corresponds to the same return step (plus a few client steps) executing the right side code to MGT.

Then with (8.11), we can prove the following by induction over the number of steps generating the event trace of \( \mathcal{H}[W_1, S_1] \).

\[
\text{If } (W_1, S_1) \preceq_{\text{MGC}} (W_2, S_2), \text{ then } \mathcal{H}[W_1, S_1] \subseteq \mathcal{H}[W_2, S_2].
\]

Also we have

\[
(\text{let } \Pi \in C_1 \ldots \Pi_n, (\sigma_{i}, \sigma_o, \otimes)) \preceq_{\text{MGC}} (\text{let } \Pi \in \text{MGC} \sigma_n, (\sigma_{i, \text{MGC}}(n), \sigma_o, \otimes)),
\]

thus we are done.

Then Lemma 47 is immediate by unfolding the definitions of \( \preceq_{\text{P}} \Gamma \) and \( \preceq_{\text{in}} \Gamma \), and applying Lemma 50.

**Proof of Lemma 48**

1. \( \Pi \preceq_{\text{P}} \Gamma \implies \Pi \preceq_{\text{P}} \Gamma \):

To prove this direction, we show that any history generated by MGC \( n \) is “equivalent” to an observable trace generated by MGCp \( n \), i.e., the following lemma holds.

**Lemma 51.** For any \( n, \sigma_o \) and \( \sigma_{\text{MGC}n}(n) \), both the following holds.

(a) For any \( T_1, T \) such that \( T_1 \in \mathcal{H}(\text{let } \Pi \in \text{MGC} \sigma_n, (\sigma_{i, \text{MGC}}(n), \sigma_o, \otimes))) \),

there exists \( T_2 \) such that \( T_2 \in \mathcal{O}(\text{let } \Pi \in \text{MGC} \sigma_n, (\sigma_{i, \text{MGC}}(n), \sigma_o, \otimes)) \) and \( T_1 \approx T_2 \).

(b) For any \( T_2, T \) such that \( T_2 \in \mathcal{O}(\text{let } \Pi \in \text{MGC} \sigma_n, (\sigma_{i, \text{MGC}}(n), \sigma_o, \otimes))) \),

there exists \( T_1 \) such that \( T_1 \in \mathcal{H}(\text{let } \Pi \in \text{MGC} \sigma_n, (\sigma_{i, \text{MGC}}(n), \sigma_o, \otimes)) \) and \( T_1 \approx T_2 \).

Here \( T_1 \approx T_2 \) is inductively defined as follows.

\[
\epsilon \approx \epsilon \quad e_1 \approx e_2 \quad T_1 \approx T_2 \quad e_1 :: T_1 \approx e_2 :: T_2.
\]

\[
(t, f_i, n) \approx (t, \text{out} \ (i, n)) \quad (t, \text{ret} \ n) \approx (t, \text{out} \ n) \quad (t, \text{obj} \ abort) \approx (t, \text{obj} \ abort)
\]

**Proof.** By constructing simulations between MGC \( n \) and MGCp \( n \).

2. \( \Pi \preceq_{\text{P}} \Gamma \implies \Pi \preceq_{\text{P}} \Gamma \):

**Proof.** For any \( n, C_1, \ldots, C_n, \sigma_o, \sigma_{\text{MGC}n} \) and \( \sigma_o \), by Lemma 50 we know

\[
\mathcal{H}(\text{let } \Pi \in C_1 \ldots \Pi_n, (\sigma_{i, \text{MGC}}(n), \sigma_o, \otimes)) \subseteq \mathcal{H}(\text{let } \Pi \in \text{MGC} \sigma_n, (\sigma_{i, \text{MGC}}(n), \sigma_o, \otimes)).
\]

Since \( \Pi \preceq_{\text{P}} \Gamma \), we know for any \( \sigma_n \) such that \( (\sigma_o, \sigma_n) \models P \), we have

\[
\mathcal{H}(\text{let } \Pi \in \text{MGC} \sigma_n, (\sigma_{i, \text{MGC}}(n), \sigma_o, \otimes)) \subseteq \mathcal{H}(\text{let } \Gamma \in \text{MGC} \sigma_n, (\sigma_{i, \text{MGC}}(n), \sigma_o, \otimes)).
\]

Thus, for any \( T \) such that

\[
T \in \mathcal{T}(\text{let } \Pi \in C_1 \ldots \Pi_n, (\sigma_{i, \text{MGC}}(n), \sigma_o, \otimes)),
\]

there exists \( T_{\text{MGC}} \) such that \( \text{get_hist}(T) = \text{get_hist}(T_{\text{MGC}}) \) and

\[
T_{\text{MGC}} \in \mathcal{T}(\text{let } \Gamma \in \text{MGC} \sigma_n, (\sigma_{i, \text{MGC}}(n), \sigma_o, \otimes)).
\]

Intuitively, we can then replace the object events in \( T \) with those in \( T_{\text{MGC}} \) and keep other events (and the order between them) unchanged. Thus the resulting trace \( T' \) satisfies \( \text{get_obsv}(T) = \text{get_obsv}(T) \). We prove \( T' \) can be produced by the corresponding abstract client program, that is

\[
T' \in \mathcal{T}(\text{let } \Gamma \in C_1 \ldots \Pi_n, (\sigma_{i, \text{MGC}}(n), \sigma_o, \otimes)).
\]

Then we are done.

Alternatively, we can actually construct a “simulation” relation \( \preceq \) (see Fig. 26(b)) between the three programs: the concrete client program, the abstract MGC and the corresponding abstract client program, and prove it satisfies the following (B.12).

For any \( W_1, S_1, W_2, S_2, W_3, S_3 \) and \( e_1 \),

if \( (W_1, S_1) \preceq (W_2, S_2; W_3, S_3) \), then
Proof of Lemma 49.

2. \( \Pi \subseteq_{=\text{MC}} \Gamma \)

(a) the program is simulated by MGC

\[
\Pi \subseteq_{=\text{MC}} \Gamma \quad \text{where } \forall i, (C_i, \kappa_i) \leq_{\text{MC}} (C'_i, \kappa'_i)
\]

\[
\Pi \subseteq_{\text{MC}} (\text{MGT}_{\text{c}}, o) \quad \text{(c, (c, x, \Gamma))} \leq_{\text{MC}} (C, (c, z_1, (\text{skip}, \text{MGT}_{\text{c}})))
\]

(b) the concrete program is “simulated” by the abstract MGC and the abstract program

Fig. 26. Simulations between programs and MGC.

(1) if \((W_1, S_1) \xrightarrow{e_1} \text{abort}\), then there exists \(T_3\) such that
\[
(W_3, S_3) \xrightarrow{T_3} \text{abort} \quad \text{and } e_1 = \text{get\_obs}(T_3);
\]

(2) if \((W_1, S_1) \xrightarrow{e_1} (W'_1, S'_1)\) and \(\text{is\_c\_lt}(e_1)\), then there exist \(W_2\) and \(S_2\) such that \((W_3, S_3) \xrightarrow{T_3} (W'_2, S'_2)\) and \((W'_1, S'_1) \subseteq (W_2, S_2 \circ \cdots \circ W_3, S_3)\).

(3) if \((W_1, S_1) \xrightarrow{e_1} (W'_1, S'_1)\) and \(\text{is\_o\_bj}(e_1)\), then there exist \(T_2, W'_2, S'_2, T_3, W'_3, S'_3\) and \(\sigma\) such that \((W_2, S_2, T_2, W'_2, S'_2, T_3, W'_3, S'_3, \sigma)\),
\[
\text{get\_hist}(e_1) = \text{get\_hist}(T_2), \quad \text{get\_obj}(T_2) = T_3 \quad \text{and } (W'_1, S'_1) \subseteq (W'_2, S'_2 \circ \cdots \circ W'_3, S'_3, \sigma)\).
\]

That is, the executions of the abstract program \(W_3\) follow the concrete program \(W_1\) for client steps and the abstract MCG \(W_2\) for object steps. Here we use \(\text{get\_obj}(T)\) to get the sub-trace of \(T\) consisting of object events only. We can prove the following from \(\text{B.12}\).

If \((W_1, S_1) \subseteq (W_2, S_2 \circ \cdots \circ W_3, S_3)\), then \(O[[W_1, S_1]] \subseteq O[[W_3, S_3]]\).

Initially,

\[
\Pi \xrightarrow{\text{MC}} \Gamma \quad \text{if } \Pi \subseteq_{\text{MC}} \Gamma \quad \text{holds, and we are done.}
\]

\(\square\)

\(\text{Proof of Lemma 49}\)

1. \(\Pi \subseteq_{\text{MC}} \Gamma \quad \iff \quad \Pi \subseteq_{\text{MC}} \Gamma \)

The premise \(\Pi \subseteq_{\text{MC}} \Gamma \) tells us that every history generated by using the concrete object \(\Pi\) can also be generated with the abstract object \(\Gamma\). Thus, to prove the concrete object \(\Pi\) is linearizable, we only need to show the abstract object \(\Gamma\) (whose methods are atomic) is linearizable with respect to itself.

**Lemma 52** (\(\Gamma\) is linearizable).

For any \(n, \sigma_{\text{MC}(n)}, \sigma_a\), and \(T\), if \(T \in \mathcal{H}[[\text{let } \Gamma \text{ in } \text{MGC}_n, (\sigma_{\text{MC}(n)}, \sigma_a, \odot)]]\), then there exist \(T_c\) and \(T'\) such that \(T_c \in \text{completions}(T)\), \(T \preceq_{\text{lin}} T'\), \(\text{seq}(T')\) and \(T' \in \mathcal{H}[[\text{let } \Gamma \text{ in } \text{MGC}_n, (\sigma_{\text{MC}(n)}, \sigma_a, \odot)]]\).

**Proof.** From the execution that generates \(T\), we construct another execution as follows. We postpone every invocation step and advance the latter return step to the single step of the atomic method body in between. We prove the resulting execution generates a history \(T'\) satisfying all the requirements.

2. \(\Pi \subseteq_{\text{MC}} \Gamma \quad \iff \quad \Pi \subseteq_{\text{MC}} \Gamma \)

The key is to prove that every linearizable history can be generated by the MGC with the abstract object \(\Gamma\).

**Lemma 53** (Rearrangement).

For any \(n, \sigma_{\text{MC}(n)}, \sigma_a, T\) and \(T'\), if \(T \preceq_{\text{lin}} T', \text{seq}(T')\) and \(T' \in \mathcal{H}[[\text{let } \Gamma \text{ in } \text{MGC}_n, (\sigma_{\text{MC}(n)}, \sigma_a, \odot)]]\), then \(T \in \mathcal{H}[[\text{let } \Gamma \text{ in } \text{MGC}_n, (\sigma_{\text{MC}(n)}, \sigma_a, \odot)]]\).
Proof. We construct the execution generating $T$, where the order to execute the atomic method body for concurrent method calls simply follows the order of the pairs of invocation and subsequent return events in $T'$. The detailed proof is by induction over the length of $T$. □

B.5.3 Equivalence for starvation-freedom

By Theorem 42 the goal is reduced to the following:

$$\Pi \subseteq^\text{fin}_P \Gamma \land \text{starvation-free}_P(\Pi) \iff \Pi \subseteq_P \Gamma.$$ 

We first define the MGC version of starvation-freedom.

**Definition 54.** SF-MGC$_P$(II), if

$$\forall n, \sigma, o, T \in \mathcal{T} \omega [\{\text{let } \Pi \text{ in } MGC_{P1_n}, (\sigma_{MGC(n)}, \sigma, \otimes)\}] \land \text{fair}(T) \land ((\sigma, \omega) \models P) \implies \text{prog-t}(T) \lor \text{abt}(T).$$

We only need to prove the following lemmas.

**Lemma 55.** $\Pi \subseteq_P \Gamma \iff \Pi \subseteq^\text{fin}_P \Gamma$.

**Lemma 56.** $\Pi \subseteq_P \Gamma \iff \text{SF-MGC}_P(\Pi)$.

**Lemma 57.** SF-MGC$_P$(II) $\implies$ starvation-free$_P$(II).

**Lemma 58.** $\Pi \subseteq^\text{fin}_P \Gamma \land \text{starvation-free}_P(\Pi) \iff \Pi \subseteq_P \Gamma$.

**Proof of Lemma 55.** For any $n, C_1, \ldots, C_n, \sigma, \sigma_o$ and $\sigma_a$ such that $(\sigma_o, \sigma_a) \models P$, for any $T$, if

$$T \in \mathcal{O}[\{\text{let } \Pi \text{ in } C_1 \| \ldots \| C_n, (\sigma_c, \sigma_o, \otimes)\}],$$

we know there exists $T_i$ such that $T = \text{get-obs}(T_i)$ and

$$T_i \in \mathcal{T}[\{\text{let } \Pi \text{ in } C_1 \| \ldots \| C_n, (\sigma_c, \sigma_o, \otimes)\}].$$

We can prove: there exists $T'_i$ and $T''_i$ such that

$$T'_i = (\text{spawn } n) : T_i \lor \text{fair}(T'_i) \land T''_i \in \mathcal{T} \omega [\{\text{let } \Pi \text{ in } C_1 \| \ldots \| C_n, (\sigma_c, \sigma_o, \otimes)\}].$$

Since $\Pi \subseteq_P \Gamma$, we know there exists $T''_i$ such that

$$T''_i \in \mathcal{T} \omega [\{\text{let } \Pi \text{ in } C_1 \| \ldots \| C_n, (\sigma_c, \sigma_o, \otimes)\}], \text{ fair}(T''_i) \land \text{get-obs}(T''_i) = \text{get-obs}(T'_i) = T : \text{get-obs}(T_i).$$

Thus there exists $T_2$ such that

$$T_2 \in \mathcal{T}[\{\text{let } \Gamma \text{ in } C_1 \| \ldots \| C_n, (\sigma_c, \sigma_o, \otimes)\}] \land \text{get-obs}(T_2) = T.$$ 

Thus $T \in \mathcal{O}[\{\text{let } \Gamma \text{ in } C_1 \| \ldots \| C_n, (\sigma_c, \sigma_o, \otimes)\}]$ and we are done.

**Proof of Lemma 56.** For any $n, \sigma_o, \sigma_a$ and $T$ such that $T \in \mathcal{T} \omega [\{\text{let } \Pi \text{ in } MGC_{P1_n}, (\sigma_{MGC(n)}, \sigma_o, \otimes)\}], \text{fair}(T)$ and $(\sigma_o, \sigma_a) \models P$, suppose

$$\neg \text{abt}(T),$$

then by the operational semantics, we only need to prove:

for any $i$ and $e$, if $e \in \text{pend-iv}(T(1..i))$, then there exists $j > i$ such that $\text{match}(e, T(j))$.

Suppose it does not hold. Then we know there exists $t_0$ such that

$$\exists i. \forall j. j \geq i \Rightarrow (T|_{t_0})(j) = (t_0, \text{obj}).$$

Thus we have

$$||(\text{get-obs}(T)|_{t_0}) < i \text{ and last}(\text{get-obs}(T)|_{t_0}) = (t_0, \text{out}, 0).$$

Since $\Pi \subseteq_P \Gamma$, we know:

$$\mathcal{O}_\omega [\{\text{let } \Pi \text{ in } MGC_{P1_n}, (\sigma_{MGC(n)}, \sigma_o, \otimes)\}] \subseteq \mathcal{O}_\omega [\{\text{let } \Gamma \text{ in } MGC_{P1_n}, (\sigma_{MGC(n)}, \sigma_o, \otimes)\}].$$

Thus there exists $T'$ such that

$$T' \in \mathcal{T} \omega [\{\text{let } \Gamma \text{ in } MGC_{P1_n}, (\sigma_{MGC(n)}, \sigma_o, \otimes)\}], \text{ fair}(T') \land \text{get-obs}(T') = \text{get-obs}(T).$$

Thus last$(\text{get-obs}(T'))|_{t_0} = (t_0, \text{out}, 0)$. But this is impossible from the operational semantics, fair$(T')$ and the fact that $\Gamma$ is atomic and total. Thus we are done.

**Proof of Lemma 57.** Similar to the proof of Lemma 56, we first prove the following two lemmas (Lemma 59 and Lemma 60). Then, from SF-MGC$_P$(II), we know: if $T' \in \mathcal{T} \omega [\{\text{let } \Pi \text{ in } MGC_{P1_n}, (\sigma_{MGC(n)}, \sigma_o, \otimes)\}, (\sigma_o, \omega) \models P$ and fair$(T')$, then either prog-t$(T')$ or abt$(T')$. By the definition of MGC$_P$, and the operational semantics, we know either prog-t$(T')$ or abt$(T')$. From Lemmas 59 and 60 we know: if $T' \in \mathcal{T} \omega [\{\text{let } \Pi \text{ in } C_1 \| \ldots \| C_n, (\sigma_o, \sigma, \otimes)\}], (\sigma_o, \omega) \models P$ and fair$(T)$, then either clt_abt$(T)$, or prog-t$(T)$, or obj_abt$(T)$. Thus we have starvation-free$_P(\Pi)$.

**Lemma 59 (MGC is the most general).** For any $n, C_1, \ldots, C_n, \sigma_c, \sigma_o$ and $T$, if
With (B.14), we can prove the following (B.15) by induction over the length of T.

Also we can prove: for any n, C₁, . . . , Cₙ, σₙ, σ₀ and σ₀ such that (σₙ, σ₀) P, then

\( T' \in \mathcal{T}_w[(\text{let } \Pi \text{ in } C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma_o, \odot)] \) and fair(T).

Here get_obj(T) gets the sub-trace consisting of only object events (i.e., method invocation, return, object fault and normal object events).

Proof. We construct a simulation relation between the arbitrary client program and the MGC, under the fixed scheduling T for the client at the left side.

Lemma 60. For any T and T', if get_obj(T) = get_obj(T') and prog-t(T'), then prog-t(T).

Proof. By unfolding the definitions.

**Proof of Lemma 58.** The key is to show the following (B.13).

For any n, C₁, . . . , Cₙ, σₙ, σ₀ and σ₀ such that (σₙ, σ₀) P.

if \( (\text{let } \Pi \text{ in } C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma_o, \odot)) \) \( \stackrel{T_1}{\longrightarrow} \omega \) and fair(T), then there exists Tₙ such that

\( (\text{let } \Gamma \text{ in } C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma_o, \odot)) \stackrel{T_n}{\longrightarrow} \omega \cdot \), get_obj(T) = get_obj(Tₙ) and fair(Tₙ).

(B.13)

Its proof is similar to the proof of \( \Pi \leq \rho \Gamma \Rightarrow \Pi \leq \rho' \Gamma \). We can replace the object events in the concrete infinite trace T with those generated by MGC using \( \Gamma \). The resulting trace Tₙ satisfies get_obj(Tₙ) = get_obj(T). From starvation-free \( \rho(\Pi) \), we can show Tₙ must be infinite and fair.

In detail, we construct a “simulation” relation \( \preceq \) (see Fig. 20b)) between the three programs: the concrete client program, the abstract MGC and the corresponding abstract client program, and prove it satisfies the following (B.14). (It is derived from (B.12).) Here we use T \( \setminus (\omega, \text{obj}) \) for a sub-trace resulting from removing all the events of the form (\( \omega, \text{obj} \)) in T.

For any W₁, S₁, W₂, S₂, W₃, S₃ and e₁, if \( (W₁, S₁) \preceq (W₂, S₂; W₃, S₃), (W₁, S₁) \), then there exist T₂, W₂, S₂, W₃ and S₃ such that

\( (W₂, S₂) \stackrel{T_2}{\longrightarrow} \omega \cdot (W₃, S₃), (W₁, S₁) \). T₃ \( \setminus (\omega, \text{obj}) \) = e₁ \( \setminus (\omega, \text{obj}) \) and \( (W₁, S₁) \preceq (W₂, S₂; W₃, S₃) \).

(B.14)

With (B.14), we can prove the following (B.15) by induction over the length of T₁.

For any W₁, S₁, W₂, S₂, W₃, S₃ and T₁, if \( (W₁, S₁) \preceq (W₂, S₂; W₃, S₃), (W₁, S₁) \), then there exist T₂, W₂, S₂, W₃ and S₃ such that

\( (W₂, S₂) \stackrel{T_2}{\longrightarrow} \omega \cdot (W₃, S₃), (W₁, S₁) \). T₃ \( \setminus (\omega, \text{obj}) \) = (W₁, S₁) \preceq (W₂, S₂; W₃, S₃).

(B.15)

Then we prove the following (B.16) by co-induction.

For any n, C₁, . . . , Cₙ, σₙ, σ₀, W₁, S₁, W₂, S₂, W₃, S₃, T₀ and T₁, if \( (\text{let } \Pi \text{ in } C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma_o, \odot)) \) \( \stackrel{T_0}{\longrightarrow} \omega \cdot (W₁, S₁) \), then there exists T₃ such that \( (W₃, S₃) \stackrel{T₃}{\longrightarrow} \omega \cdot \) and T₃ \( \setminus (\omega, \text{obj}) \).

(B.16)

Also we can prove: for any n, C₁, . . . , Cₙ, σₙ, σ₀ and σ₀ such that (σₙ, σ₀) P, then

\( (\text{let } \Pi \text{ in } C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma_o, \odot)) \) \( \preceq \) (let \( \Gamma \) in MGCn, (\( \sigma_{\text{MGC}(n)} \), σ₀, \odot)); |let \( \Gamma \) in C₁ \parallel \ldots \parallel C_n, (\sigma_c, \sigma_o, \odot)|.

If \( (\text{let } \Pi \text{ in } C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma_o, \odot)) \) \( \stackrel{T}{\longrightarrow} \omega \cdot \) and \( \text{fair}(T) \), by starvation-free \( \rho(\Pi) \), we know prog-t(T). Then from (B.16) we get: there exists Tₙ such that

\( (\text{let } \Gamma \text{ in } C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma_o, \odot)) \) \( \stackrel{T_n}{\longrightarrow} \omega \cdot \), and Tₙ \( \setminus (\omega, \text{obj}) \) = Tₙ \( \setminus (\omega, \text{obj}) \).

Thus we know get_obj(T) = get_obj(Tₙ).

Below we prove fair(Tₙ). Since fair(T) and |T| = \( \omega \), we know for any t,

\[ |T_0| = \omega \] and last(Tₙ) = (t, term) .
We first define the following contextual refinement $\hat{\Gamma}$ which assumes fair scheduling at the concrete side only.

**Definition 61.** $\Pi \hat{\sqsubseteq}_P \Gamma$ if

$$\forall n, C_1, \ldots, C_n, \sigma, \Sigma. ((\sigma, \Sigma) \models P) \implies O_{\omega} [[(\text{let } \Pi \text{ in } C_1 \parallel \ldots \parallel C_n), (\sigma, \Sigma, \circ \circ)] \subseteq O_{\omega} [[(\text{let } \Gamma \text{ in } C_1 \parallel \ldots \parallel C_n), (\sigma, \Sigma, \circ \circ)]].$$

We also define the assertion $id$ for the identity relation between the lower-level and the higher-level states:

$$(\sigma, \Sigma) \models id \iff \sigma = \Sigma$$

Then $wr_1(id)$ adds the shared variable $1$ to the higher-level states, and $wr_1^{-1}(id)$ removes $1$ which should be $0$:

$$(\sigma, \Sigma) \models wr_1(id) \iff \Sigma = \sigma \cup \{1 \leadsto 0\}$$

$$(\sigma, \Sigma) \models wr_1^{-1}(id) \iff \Sigma = \sigma \cup \{1 \leadsto 0\}$$

To prove Theorem 44 we prove the following:

$$\Pi \sqsubseteq_{\text{fin}}^P \Gamma \land \text{deadlock-free}_{P}(\Pi) \iff \Pi \hat{\sqsubseteq}_P \Gamma \quad \text{(B.17)}$$

$$\Pi \sqsubseteq_{\text{wr}(P)} \hat{\sqsubseteq}_P \Gamma \implies \Pi \sqsubseteq_{\text{fin}}^P \Gamma \quad \text{(B.18)}$$

$$\Pi \sqsubseteq_{\text{fin}}^P \Gamma \land \text{deadlock-free}_{P}(\Pi) \implies \Pi \sqsubseteq_{\text{wr}(P)} \hat{\sqsubseteq}_P \Gamma \quad \text{(B.19)}$$

**Proofs of (B.17).** We first define the MGC version of deadlock-freedom.

**Definition 62.** $\text{deadlock-free}^P_{MGC}(\Pi)$, if

$$\forall n, \sigma, T. T \in T_o [[(\text{let } \Pi \text{ in } \text{MGC}_{P_1}), (\sigma_{MGC(n)}, \sigma, \circ \circ)] \land \text{fair}(T) \land ((\sigma, \circ \circ) \models P) \implies \text{prog-s}(T) \lor \text{abt}(T)].$$

We only need to prove the following lemmas.

**Lemma 63.** $\Pi \hat{\sqsubseteq}_P \Gamma \implies \Pi \sqsubseteq_{\text{fin}}^P \Gamma$.

**Lemma 64.** $\Pi \hat{\sqsubseteq}_P \Gamma \implies \text{deadlock-free}^P_{MGC}(\Pi)$.

**Lemma 65.** $\text{deadlock-free}^P_{MGC}(\Pi) \implies \text{deadlock-free}_{P}(\Pi)$.

**Lemma 66.** $\Pi \sqsubseteq_{\text{fin}}^P \Gamma \land \text{deadlock-free}_{P}(\Pi) \implies \Pi \hat{\sqsubseteq}_P \Gamma$.

**Proof of Lemma 63.** Similar to the proof of Lemma 55.

**Proof of Lemma 64.** Similar to the proof of Lemma 56. For any $n, \sigma_o, \sigma_a$ and $T$ such that $T \in T_o [[(\text{let } \Pi \text{ in } \text{MGC}_{P_1}), (\sigma_{MGC(n)}, \sigma_o, \circ \circ)], \text{fair}(T)$ and $(\sigma_o, \sigma_a) \models P$, suppose

$$\neg \text{abt}(T),$$

then by the operational semantics, we only need to prove:

for any $i$ and $e$, if $e \in \text{pend}_{\text{inv}}(T(1..i))$, then there exists $j > i$ such that is_ret$(T(j))$ holds.

Suppose it does not hold. Then we know:
Since \( \text{fair}(T) \), we have
\[
\forall t_0 \in [1..n]. \text{last(get}_\text{obsv}(T)|_{t_0}) = (t_0, \text{out}, 0) \lor \text{last(get}_\text{obsv}(T)|_{t_0}) = (t_0, \text{out}, 2) .
\]
Since \( \Pi \subseteq_{P} \Gamma \), we know:
\[
O_{\omega}[(\{\text{let } \Pi \text{ in } \text{MGCp}_1 \}, (\sigma_{\text{MGC}(|\Pi|)}, \sigma_0, \odot)) \subseteq O_{\omega}[(\{\text{let } \Gamma \text{ in } \text{MGCp}_1 \}, (\sigma_{\text{MGC}(|\Pi|)}, \sigma_0, \odot))].
\]
Thus there exists \( T' \) such that
\[
T' \in \mathcal{T}_\omega[(\{\text{let } \Gamma \text{ in } \text{MGCp}_1 \}, (\sigma_{\text{MGC}(|\Pi|)}, \sigma_0, \odot)) \text{ and } \text{get}_\text{obsv}(T') = \text{get}_\text{obsv}(T).
\]
Thus \( \forall t_0 \in [1..n]. \text{last(get}_\text{obsv}(T')|_{t_0}) = (t_0, \text{out}, 0) \lor \text{last(get}_\text{obsv}(T')|_{t_0}) = (t_0, \text{out}, 2) . \) But this is impossible from the operational semantics and the fact that \( \Gamma \) is atomic and total. Thus we are done. 

**Proof of Lemma 67.** Similar to the proof of Lemma 57. From deadlock-free\( _P^{\text{MGC}}(\Pi) \), we get: if \( T' \in \mathcal{T}_\omega[(\{\text{let } \Pi \text{ in } \text{MGCp}_1 \}, (\sigma_{\text{MGC}(|\Pi|)}, \sigma_0, \odot))], \text{ and } \text{fair}(T'), \) then either prog-s(\( T' \)) or obj_abt(\( T' \)). By the definition of MGCp and the operational semantics, we know either prog-s(\( T' \)) or obj_abt(\( T' \)). From Lemma 59, we can prove: if \( T' \in \mathcal{T}_\omega[(\{\text{let } \Pi \text{ in } C_1 \mid \ldots \mid C_n \}, (\sigma_c, \sigma_0, \odot))], \text{ and } \text{fair}(T), \) then either ct_abt(\( T \)), or prog-s(\( T \), or obj_abt(\( T \). Thus we have deadlock-free\( _P^{\text{MGC}}(\Pi) \). 

**Proofs of Lemma 66.** Similar to the proof of Lemma 58. The key is to show the following (B.20).

For any \( n, C_1, \ldots, C_n, \sigma_c, \sigma_o, \) and \( \sigma_a \) such that \((\sigma_c, \sigma_o) \models P\),
if \((\{\text{let } \Pi \text{ in } C_1 \mid \ldots \mid C_n \}, (\sigma_c, \sigma_o, \odot)) \xrightarrow{T} \omega \text{ and } \text{fair}(T)\), then there exists \( T_a \) such that
\[
(\{\text{let } \Gamma \text{ in } C_1 \mid \ldots \mid C_n \}, (\sigma_c, \sigma_o, \odot)) \xrightarrow{T_a} \omega \text{ and } \text{get}_\text{obsv}(T_a) = \text{get}_\text{obsv}(T).
\]

As in the proofs for (B.13), we construct a “simulation” relation \( \preceq \) (see Fig. 26 b)) between the three programs: the concrete client program, the abstract MGC and the corresponding abstract client program, and prove it satisfies the following (B.21).

For any \( n, C_1, \ldots, C_n, \sigma_c, \sigma_o, \) and \( \sigma_a \) such that \((\sigma_c, \sigma_o) \models P\),
if \((\{\text{let } \Pi \text{ in } C_1 \mid \ldots \mid C_n \}, (\sigma_c, \sigma_o, \odot)) \xrightarrow{T} \omega \text{ and } \text{fair}(T)\), then there exists \( T_a \) such that \((\omega, \text{obj}) \models \text{get}_\text{obsv}(T_a)\).

Also we can prove: for any \( n, C_1, \ldots, C_n, \sigma_c, \sigma_o, \) and \( \sigma_a \) such that \((\sigma_c, \sigma_o) \models P\),
if \((\{\text{let } \Pi \text{ in } C_1 \mid \ldots \mid C_n \}, (\sigma_c, \sigma_o, \odot)) \xrightarrow{T} \omega \text{ and } \text{fair}(T)\),
then there exists \( T_a \) such that \((\omega, \text{obj}) \models \text{get}_\text{obsv}(T_a)\) and we are done.

**Proofs of (B.18).** We only need to prove the following:
\[
\text{wr}_1(\Gamma) \xrightarrow{\text{wr}_1^{-1}(\omega)} \text{wr}_2(\Gamma).
\]

By (B.17), we only need to show:
\[
\text{wr}_1(\Gamma) \xrightarrow{\text{fin}(\omega)} \text{wr}_1^{-1}(\omega) \text{wr}_2(\Gamma) \xrightarrow{\text{deadlock-free}_{\text{wr}_1^{-1}(\omega)}(\text{wr}_2(\Gamma))}.
\]

**Proof of (B.22).** We want to show: for any \( n, C_1, \ldots, C_n, \sigma_c \) and \( \sigma_a \),
\[
O[(\{\text{let } \text{wr}_1(\Gamma) \text{ in } C_1 \mid \ldots \mid C_n \}, (\sigma_c, \sigma_a \cup \{1 \Rightarrow 0\}))] \subseteq O[(\{\text{let } \Gamma \text{ in } C_1 \mid \ldots \mid C_n \}, (\sigma_c, \sigma_a))].
\]
We construct a simulation relation \( \preceq \) as follows.
(let \(w_\tau(\Gamma)\) in \(C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma_o \equiv \{1 \sim \ldots \}, \odot)) \leq (let \Gamma in C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma_o \equiv \{1 \sim \ldots \}, n \sim \ldots \})
\[ (C, \sigma_o \equiv \{1 \sim \}, o) \leq (C, \sigma_o, o) \]
\(w_\tau((C)); w_\tau(\text{return } E)\); noret, \(\sigma_o \equiv \{1 \sim \}, k'\) \(\leq (C); \text{return } E\); noret, \(\sigma_o, k\)
\(w_\tau((C)); w_\tau(\text{return } E)\); noret, \(\sigma_o \equiv \{1 \sim \}, k'\) \(\leq (C); \text{return } E\); noret, \(\sigma_o, k\)
\(w_\tau((C)); w_\tau(\text{return } E)\); noret, \(\sigma_o \equiv \{1 \sim \}, k'\) \(\leq (C); \text{return } E\); noret, \(\sigma_o, k\)
\(w_\tau((C)); w_\tau(\text{return } E)\); noret, \(\sigma_o \equiv \{1 \sim \}, k'\) \(\leq (C); \text{return } E\); noret, \(\sigma_o, k\)
\(w_\tau((C)); w_\tau(\text{return } E)\); noret, \(\sigma_o \equiv \{1 \sim \}, k'\) \(\leq (C); \text{return } E\); noret, \(\sigma_o, k\)
\(w_\tau((C)); w_\tau(\text{return } E)\); noret, \(\sigma_o \equiv \{1 \sim \}, k'\) \(\leq (C); \text{return } E\); noret, \(\sigma_o, k\)
\(w_\tau((C)); w_\tau(\text{return } E)\); noret, \(\sigma_o \equiv \{1 \sim \}, k'\) \(\leq (C); \text{return } E\); noret, \(\sigma_o, k\)
\(w_\tau((C)); w_\tau(\text{return } E)\); noret, \(\sigma_o \equiv \{1 \sim \}, k'\) \(\leq (C); \text{return } E\); noret, \(\sigma_o, k\)

By case analysis and operational semantics, we can prove the following property of the simulation.

If \((W_B, S_B) \leq (W_A, S_A)\), then the following hold.

1. If \((W_B, S_B) \rightarrow_B^* \text{ abort}\), then there exists \(T_A\) such that \((W_A, S_A) \rightarrow_A^* \text{ abort}\) and \(e_B = \text{get_observ}(T_A)\).
2. If \((W_B, S_B) \rightarrow_B^* (W_B', S_B')\), then there exist \(T_A, W_A'\) and \(S_A'\) such that \((W_A, S_A) \rightarrow_A^* (W_A', S_A')\), \(\text{get_observ}(e_B) = \text{get_observ}(T_A)\) and \((W_B', S_B') \leq (W_A', S_A')\).

With (B.24), we can prove the following by induction over the number of steps generating the event trace of \(\mathcal{O}[W_B, S_B]\).

If \((W_B, S_B) \leq (W_A, S_A)\) and \(T \in \mathcal{O}[W_B, S_B]\), then \(T \in \mathcal{O}[W_A, S_A]\).

Since
\[(\text{let } w_\tau(\Gamma) \text{ in } C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma_o \equiv \{1 \sim 0\}, \odot)) \leq (\text{let } \Gamma \text{ in } C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma_o, \odot))\],
from (B.25), we are done.

Proof of (B.22). We want to show: for any \(n, C_1, \ldots, C_n, \sigma_c\) and \(\sigma_o\),
\[\forall T. T \in T_n[\text{let } w_\tau(\Gamma) \text{ in } C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma_o)] \land (\sigma_o(1) = 0) \implies \text{deadlock-free}(T)\].

It is reduced to the following.
\[\forall T. T \in T_n[\text{let } w_\tau(\Gamma) \text{ in } C_1 \parallel \ldots \parallel C_n, (\sigma_c, \sigma_o)] \land (\sigma_o(1) = 0) \land \text{fair}(T) \land \neg \text{abt}(T) \implies \text{prog-s}(T)\].

- If \(|T| \neq \omega\), we know \(\text{prog-s}(T)\) must hold.
- If \(|T| = \omega\), suppose \(\text{prog-s}(T)\) does not hold. Then there exist \(i\) and \(e\) such that \(e \in \text{pend_inv}(T(1..))\) and \(\forall j, j > i \implies \neg \text{is_ret}(T(j))\).

Suppose the execution generating such \(T\) is:
\[(W_0, S_0) \rightarrow^{(1)} (W_1, S_1) \rightarrow^{(2)} (W_2, S_2) \rightarrow^{(3)} \ldots\]

Here we write \(W_0\) for \(\text{let } w_\tau(\Gamma) \text{ in } C_1 \parallel \ldots \parallel C_n\) and \(S_0\) for \((\sigma_c, \sigma_o, \odot)\).

Suppose \(t_i(e) = t\) and \(e = T(t_0)\). We know \(\exists j, j > t_0 \land t_j(e) = T(j))\).

Since \(\text{fair}(T)\), we know \(|T(t)| = \omega\). From the operational semantics, we know there exist \(i_1, i_2, \ldots\) (an infinite number) such that \(0 < i_1 < i_2 < \ldots\) \(T(i_1) = T(i_2) = \ldots = (t, \text{obj})\), and the code of thread \(t\) in \(W_{i_1-1}, W_{i_2-1}, W_{i_2-1}, \ldots\) is the same, which is either \(w_\tau((C)); w_\tau(\text{return } E)\); noret or \(w_\tau(\text{return } E)\); noret.

Suppose the lock \(l\) has the values \(t_1, t_2, \ldots\) in the states \(S_{i_1}, S_{i_2}, \ldots\). We know \(t_j \neq 0, t_j \neq 0, \ldots\). Since there are only \(n\) threads, we know there exists \(j_1, j_2, \ldots\) (an infinite number) such that \(t_{j_1} = t_{j_2} = \ldots\). In other words, thread \(t_j\) holds the lock infinitely often in the execution.

Since \(\text{fair}(T)\), we know \(|T(t_j)| = \omega\). By the operational semantics, we know \(t_j\) must return infinitely often.

So we get a contradiction. Thus \(\text{prog-s}(T)\) holds.

Thus we are done.

\[\square\]
Proofs of $\text{(B.19)}$. The key is to show the following $\text{(B.26)}$.

For any $n, C_1, \ldots, C_n, \sigma_c, \sigma_a, \sigma_l$ and $\sigma_b$ such that $(\sigma_c, \sigma_l) \models P$ and $\sigma_l = \sigma_b \uplus \{1 \rightarrow 0\}$, if $([\text{let } \Gamma \text{ in } C_1 \| \ldots \| C_n], (\sigma_c, \sigma_b, \open)) \xrightarrow{T} \omega \cdot$ and $\text{fair}(T)$, then there exists $T_b$ such that

$$([\text{let } \Gamma \text{ in } C_1 \| \ldots \| C_n], (\sigma_c, \sigma_a, \open)) \xrightarrow{T_b} \omega \cdot, \text{fair}(T_b) \text{ and } \text{get\_obs}(T) = \text{get\_obs}(T_b).$$

(B.26)

Following the proof of $\text{(B.20)}$, we get:

For any $n, C_1, \ldots, C_n, \sigma_c, \sigma_a$ and $\sigma_l$ such that $(\sigma_c, \sigma_l) \models P$, if $([\text{let } \Gamma \text{ in } C_1 \| \ldots \| C_n], (\sigma_c, \sigma_b, \open)) \xrightarrow{T} \omega \cdot$ and $\text{fair}(T)$, then there exists $T_a$ such that

$$([\text{let } \Gamma \text{ in } C_1 \| \ldots \| C_n], (\sigma_c, \sigma_l, \open)) \xrightarrow{T_a} \omega \cdot \text{ and } T_a \setminus (\_ \text{ obj}) = T_b \setminus (\_ \text{ obj}).$$

(B.27)

Since $\text{fair}(T)$, we can prove that $T_a$ satisfies the following $\text{cltfair}$ in Definition $\text{(B.7)}$. Since $T \setminus (\_ \text{ obj}) = T_b \setminus (\_ \text{ obj})$, from $\text{prog\_s}(T)$, we know $\text{prog\_s}(T_b)$ holds. By the following Lemma $\text{(B.68)}$, we can construct the execution of $[\text{let } \Gamma \text{ in } C_1 \| \ldots \| C_n]$ generating the trace $T_b$ such that $\text{fair}(T_b)$ and $T_a \setminus (\_ \text{ obj}) = T_b \setminus (\_ \text{ obj})$ hold. Thus we are done.

Definition 67. $\text{cltfair}(T_a)$ iff $[T_a] = \omega \implies \forall t \in [1..\text{num}(T_a)], |(T_a[t])| = \omega \lor \text{last}(T_a[t]) = (t, \text{term}) \lor \text{last}(T_a[t]) = (t, f, f, n)$

Lemma 68. If $([\text{let } \Gamma \text{ in } C_1 \| \ldots \| C_n], (\sigma_c, \sigma_b, \open)) \xrightarrow{T_a} \omega \cdot$, $\text{cltfair}(T_a)$ and $\text{prog\_s}(T_a)$, then there exists $T_b$ such that $([\text{let } \Gamma \text{ in } C_1 \| \ldots \| C_n], (\sigma_c, \sigma_a \uplus \{1 \rightarrow 0\}, \open)) \xrightarrow{T_b} \omega \cdot, \text{fair}(T_b)$ and $T_a \setminus (\_ \text{ obj}) = T_b \setminus (\_ \text{ obj})$.

Proof: We construct a simulation relation $\succeq$ that satisfies the following property.

If $T \models (W_a, S_a) \succeq (W_b, S_b)$ and $(W_a, S_a) \xrightarrow{T_a} (W_a', S_a')$, then there exist $T_b, W'_b$ and $S'_b$ such that $(W_b, S_b) \xrightarrow{T_b} (W_b', S_b')$, $T\,: e_a \models (W_a', S_a') \succeq (W_b', S_b')$. $e_a \setminus (\_ \text{ obj}) = T_b \setminus (\_ \text{ obj})$ and if $\text{is\_ret}(e_a)$ holds, then $\forall e \in \text{pend\_inv}(T) \implies \exists i. \text{tid}(e) = \text{tid}(T_b(i))$.

(B.28)

The simulation is established as follows.

$$T \models ([\text{let } \Gamma \text{ in } C_1 \| \ldots \| C_n], (\sigma_c, \sigma_a \uplus \{1 \rightarrow 0\}, \_)) \succeq ([\text{let } \Gamma \text{ in } C_1 \| \ldots \| C_n], (\sigma'_c, \sigma'_a \uplus \{1 \rightarrow 0\}, \_))$$

if $\forall i. (C_i, \sigma_i, \kappa_i) \preceq (C'_i, \sigma'_i, \kappa'_i)$

$$(C, \sigma, \open) \preceq_t (C, \sigma_a \uplus \{1 \rightarrow 0\}, \open)$$

$$(\text{return } E; \text{noret }, \sigma, \kappa) \preceq_t (\text{wr}_1(C)); \text{wr}_2(\text{return } E); \text{noret }, \sigma \uplus \{1 \rightarrow 0\}, \kappa')$$

$$(\text{return } E; \text{noret }, \sigma, \kappa) \preceq_t (\text{wr}_1(C)); \text{wr}_2(\text{return } E); \text{noret }, \sigma \uplus \{1 \rightarrow 0\}, \kappa')$$

where $\kappa' = (\kappa \uplus \{1 \rightarrow \omega \uplus 2 \rightarrow \omega\}, x, C')$, if $\kappa = (\kappa, x, C')$.

We can prove $\text{(B.28)}$ by case analysis and operational semantics. The idea is to execute the lock instructions of the threads which are pending just after the lock instruction of the thread that returns. Then if $\text{is\_ret}(e_a)$ holds, the pending threads in $T$ must have been executed in the execution of $(W_b, S_b)$.

With $\text{(B.28)}$, we can prove the following by co-induction over the event trace $T_a$ generated in $(W_a, S_a) \xrightarrow{T_a} \omega \cdot$.

If $e \models (W_a, S_a) \succeq (W_b, S_b), (W_a, S_a) \xrightarrow{T_a} \omega \cdot$ and $\text{prog\_s}(T_a)$, then there exists $T_b$ such that $(W_b, S_b) \xrightarrow{T_b} \omega \cdot, T_a \setminus (\_ \text{ obj}) = T_b \setminus (\_ \text{ obj})$ and $\forall e \in \text{pend\_inv}(T_b) \implies \forall i. \exists j > i. \text{tid}(T_b(j)) = \text{tid}(e)$.

(B.29)

Since $e \models ([\text{let } \Gamma \text{ in } C_1 \| \ldots \| C_n], (\sigma_c, \sigma_a, \open)) \succeq ([\text{let } \Gamma \text{ in } C_1 \| \ldots \| C_n], (\sigma_c, \sigma_a \uplus \{1 \rightarrow 0\}, \open))$, from $\text{(B.29)}$, we know there exists $T_b$ such that

$$([\text{let } \Gamma \text{ in } C_1 \| \ldots \| C_n], (\sigma_c, \sigma_a \uplus \{1 \rightarrow 0\}, \open)) \xrightarrow{T_b} \omega \cdot.$$

$T_a \setminus (\_ \text{ obj}) = T_b \setminus (\_ \text{ obj})$ and $\forall e \in \text{pend\_inv}(T_b) \implies \forall i. \exists j > i. \text{tid}(T_b(j)) = \text{tid}(e)$.

Since $([\text{let } \Gamma \text{ in } C_1 \| \ldots \| C_n], (\sigma_c, \sigma_a, \open)) \xrightarrow{T_b} \omega \cdot$ and $\text{cltfair}(T_b)$, we know $\text{fair}(T_b)$ holds. Thus we are done. \qed
C.  Starvation-free examples

C.1 Counter with ticket lock

Our logic LiLi can be applied to verify simple objects using a single lock to protect all the object data. As examples, we verify the counter object using a ticket lock (see below), or a queue lock (see Secs. 2.3 and 2.4), or a test-and-set lock (see Sec. 3.1).

Fig. 27 shows the concrete implementation of the counter sflInc with a ticket lock, where we write the auxiliary code in red. We implicitly assume every instruction that accesses the shared state is in an atomic block.

As we explained in Sec. 2 sflInc in Fig. 1 is linearizable with respect to the abstract atomic operation IHC because sflInc ensures mutually exclusive access to x. When a thread acquires the lock, the shared resource x at both the concrete and the abstract sides is transferred to the thread’s local state. sflInc ensures starvation-freedom because the threads which are currently requesting the lock constitute a queue.

We introduce some auxiliary structures to facilitate the proof. As shown in Fig. 27, we introduce the explicit tickets ticket0, ticket1, ... Each ticket records the ID of the unique thread which gets the ticket number i. We use cid to record the current thread ID. The explicit connection from the ticket numbers to the thread IDs gives us the knowledge about the queue of the threads requesting the lock.

Besides, each ticket i is accompanied with a waiting bit wait, to indicate whether the thread ticketi is requesting the lock or has entered its critical section. It helps describe the ownership transfer over the resource x. Initially waiti is false. It is set to true when the thread gets its ticket at line 2. After the loop, waiti is reset (line 7) to transfer the shared resource x to the thread so that the thread can freely access x in the critical section (line 8).

Fig. 28 defines the precise invariant I which determines the boundaries of shared states. A shared state contains the lock (with owner = n1 and next = n2), the wait bits (waits) and the protected resource if waitowner holds. Here resource requires x to be the same at the concrete and the abstract sides. lock(tl, n1, n2) contains the auxiliary tickets in addition to owner and next, where tl (hidden in the definition of I) is the list of the threads ticket0, ticket1, .... We also use locked(tl, n1, n2) 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 locklr(tl, n1, n2) short for lock(tl, n1, n2) ∧ (t ∉ tl). That is, the thread t is irrelevant to the lock: it does not acquire or request the lock. The precondition P is stronger than I. For thread t, P(t) requires the lock to satisfy ∃tl. locked(tl, n1, n2).

The guarantee condition G in Fig. 28 defines the atomic actions of a thread t. ReqLock∗ adds the thread t at the end of tl of the threads requesting the lock and increments next. It corresponds to line 2 of Fig. 27 AcqResource transfers the resource to the thread t when it has acquired the lock, corresponding to line 7 of Fig. 27 RelLock releases the lock and transfers the resource back to the shared state. The thread t which currently holds the lock is dequeued from the list tl and owner is incremented. It corresponds to line 9 of Fig. 27. The rely condition R include all the atomic actions made by the environment threads.

Next we define the definite action D in Fig. 28. As we explained in Sec. 2 the thread t’s definite action Dt requires that whenever t holds a lock with owner = n1 (specified by dp(t, n1)), it should eventually release the lock by incrementing owner to n1 + 1 (specified by dp(t, n1 + 1)). Note that dp(t, n1) allows the resource to be either in the shared state or in the thread t’s local state. The former case means that the thread t has not executed line 7 but its loop at lines 4–6 must terminate. Besides, dp(t, n1) is stable under the environment which may enqueue more threads into the list tl of waiting threads. We can prove the side conditions about well-formedness of specifications in the OBJ rule in Fig. 29.

Fig. 29 shows the proof outline. To verify the loop at lines 4–6, we first define J and Q in Fig. 28. Both J and Q are stronger than I. For thread t, J(t) says t is requesting the lock. Here locked(t, tl, n1, X, n2) says (1) t takes the ticket number X, which satisfies n1 = owner ≤ X < next = n2, and (2) the threads requesting the lock before and after t constitute the lists tl1 and tl2 respectively. Then, the first thread of the whole concatenated list tl1 :: tl2 holds the lock. Q specifies the case when tl1 is empty (thus X = owner). The loop terminates when Q holds. We also strengthen the guarantee of the loop to G ′ def = [I], the identity transitions. We have Sta{J, Q}, G ′ ⊇ R).

Next we define f and prove J ⇒ (R, G′: D ′ 7 → Q). The metric function f maps each shared state S to the value of (X – owner) at that state. Here X is a logical variable recording the ticket number of the current thread (i.e., X = 1 holds). Since J ensures owner ≤ X, we can use the usual order on natural numbers as the associated well-founded order. The condition J ⇒ (R, G′: D ′ 7 → Q) holds due to the following reasons:

1. Each action in R or G′ either increases owner or keeps owner unchanged.
2. Either Q holds, or owner < X and some environment thread t′ acquires the lock. In the latter case, the value of (X – owner) decreases when t′ releases the lock. That is, the metric decreases after a definite action made by the environment.

Finally we conclude the full correctness of sflInc with respect to the atomic IHC under fair scheduling. By Theorem 2 (Soundness), we get linearizability and starvation-freedom of sflInc.
initialize() { owner := 0; next := 0; } sfInc() {
  1 local i, o, r;
  2 <i := getAndInc(next); ticket₁ := cid; wait₁ := true;>
  3 o := owner;
  4 while (i != o) {
  5 o := owner;
  6 }
  7 <wait₁ := false;>
  8 r := x; x := r + 1;
  9 owner := i + 1;
}

Figure 27. Counter sfInc with a ticket lock (auxiliary code in red).

tl := ε | t::tl
list2set(ε) ≜ ∅
list2set(t::d) ≜ {t} ∪ list2set(d)

I ≜ ∃n₁,n₂. lock(n₁,n₂) ∨ (waitₙ₁ ∨ ¬waitₙ₁) ∨ waits(n₁,n₂)
resource ≜ (x = X) Sites ≜ (x = o)
waits(n₁,n₂) ≜ (∃ₖ₁<₁<₁⁺₁) ∨ (∃ₙ₂<₁<₁⁺₁ wait₁)
lock(n₁,n₂) ≜ ∀tl. lock(tl,n₁,n₂)
lock(tl,n₁,n₂) ≜ ((owner = n₁) ∨ (next = n₂) ∨ (n₁ ≤ n₂)) ∨ tickets(0,n₁) ∨ tickets(tl,n₁,n₂) ∨ ticket_new(n₁)
tickets(n₁,n₂) ≜ ∃tl. tickets(tl,n₁,n₂)
tickets(tl,n₁,n₂) ≜ (tl = e) ∧ (n₁ = n₂) ∧ emp ∨ ∃t.t′. (tl = t::t′) ∧ (t ∉ list2set(t′)) ∧ (ticketₙ₁ = t) ∨ tickets(t′,n₁ + 1,n₂)
ticket_new(n₁) ≜ (∃₁<₁⁺₁ ticket₁ = -1)

P₁ ≜ ∃n₁,n₂. lock้ง(n₁,n₂) ∨ (waitₙ₁ ∨ ¬waitₙ₁) ∨ waits(n₁,n₂)
lock้งₙ₁(n₁,n₂) ≜ ∃tl. lock้งₙ₁(tl,n₁,n₂) lock้งₙ₁(tl,n₁,n₂) ≜ lock(tl,n₁,n₂) ∨ (t ∉ list2set(tl))

Rₙ ≜ ∀t'. Gₙ Gₙ ≜ (ReqLockₙ ∨ AcqResourceₙ ∨ RelLockₙ ∨ Id) ∨ Id ∧ (I ⊨ I)

ReqLockₙ ≜ ∀tl,n₁,n₂. lock้งₙ₁(tl,n₁,n₂) ∨ (waitₙ₂) ∧ lock(tl::t,n₁,n₂ + 1) ∨ waitₙ₂
AcqResourceₙ ≜ ∀tl,n₁,n₂. (lock(t::tl,n₁,n₂) ∨ waitₙ₁) ∧ (lock(t::tl,n₁,n₂) ∨ (¬waitₙ₁))
RelLockₙ ≜ ∀tl,n₁,n₂. lock(t::tl,n₁,n₂) ∨ (lock้งₙ₁(tl,n₁ + 1,n₂) ∨ resource)

Dₙ ≜ ∀n₁. ,dₙ₁(n₁) ≜ dₙ₁(n₁)
dₙ₁(n₁) ≜ ∃tl,n₁,n₂. lock(t::tl,n₁,n₂) ∨ true ∧ I
dₙ₁(n₁) ≜ ∃tl,n₁,n₂. lock้งₙ₁(n₁ + 1,n₂) ∨ true ∧ I

Figure 28. Invariant, precondition, rely/guarantee and definite action of counter with ticket lock.
We can prove:

- \( \text{locked}_{I_{\text{inc}}}(n_1, n_2) \) is verified as follows. Let

\[
\begin{align*}
\text{inc}(): \\
1 & \quad \text{local } i, o, r; \\
2 & \quad \{ P \land \text{arem}(\text{INC}) \} \\
3 & \quad < i := \text{getAndInc}(\text{next}); \quad \text{ticket} := \text{cid}; \quad \text{wait} := \text{true}; > \\
4 & \quad \{ 3 n_1, n_2, P_0(n_1, n_2) \land (i = n) \land \text{arem}(\text{INC}) \} \\
5 & \quad \{ 3 n_1, n_2, P_0(n_1, n_2) \land (i = n) \land \text{arem}(\text{INC}) \} \\
6 & \quad o := \text{owner}; \\
7 & \quad \{ 3 n_1, n_2, P_0(n_1, n_2) \land (i = n) \land (o = n_1) \land \text{arem}(\text{INC}) \} \\
8 & \quad \{ 3 n_1, n_2, P_0(n_1, n_2) \land (i = n) \land (o = n_1) \land \text{arem}(\text{INC}) \land \neg(n - o) \} \\
9 & \quad \text{while} (1 != o) \{ \\
10 & \quad \{ P \land \text{arem}(\text{skip}) \} \\
11 & \quad \}
\end{align*}
\]

Here the loop at lines 4-6 is verified as follows. Let

\[
\begin{align*}
p' & \quad \equiv \exists n_1, n_2. (P_1(n_1, n_2) \land \neg(n - (o + 1)) \lor P_3(n_1, n_2) \land \neg(n - o)) \land (o \leq n_1 \leq n = 1) \land (o \neq 1) \land \text{arem}(\text{INC}) \\
J & \quad \equiv \exists n_1, n_2. P_0(n_1, n_2) \\
Q & \quad \equiv \text{Enabled}(D) \\
G' & \quad \equiv \{ I \} \\
f(\emptyset) & \quad = k \quad \text{iff} \quad \emptyset \models (n - \text{owner} = k)
\end{align*}
\]

\[
\begin{align*}
(3 n_1, n_2, P_0(n_1, n_2) \land (i = n) \land (o \leq n_1) \land (i \neq o) \land \text{arem}(\text{INC}) \land \neg(n - o) \land Q \land \text{true}) \\
& \quad \implies (3 n_2, P_1(n_1, n_2) \land (i = n) \land (o < n) \land \text{arem}(\text{INC}) \land \neg(n - (o + 1)) \land \neg(o(1) \land \text{emp}) \\
& \quad \{ p' \} \\
& \quad \{ \exists n_1, n_2. (P_1(n_1, n_2) \land \neg(n - (o + 1)) \lor P_3(n_1, n_2) \land \neg(n - o)) \land (o \leq n_1 \leq n = 1) \land (o \neq 1) \land \text{arem}(\text{INC}) \} \\
5 & \quad o := \text{owner}; \\
6 & \quad \{ 3 n_1, n_2, P_0(n_1, n_2) \land (i = n) \land (o \leq n_1) \land \text{arem}(\text{INC}) \land \neg(n - o) \} \\
7 & \quad \{ p \}
\end{align*}
\]

We can prove:

1. \( J \implies I; \quad Q \implies I; \quad \text{Sta}(J, G' \lor R) \).
2. \( J \implies (R, G': D \rightarrow Q) \) (where picking \( D' \) in the WHL rule as \( D \)).

Figure 29. Proof outline. We prove: \( D, R, G, I \vdash \{ P \land \text{arem}(\text{INC}) \} C \{ P \land \text{arem}(\text{skip}) \} \).
initialize(){ tail := 0; flag[0] := true; flag[1..TNUM-1] := false; }

inc()
1 local i, b, r;
2 <i := getAndInc(tail) % TNUM; queue[i] := cid; wait[i] := true>;
3 b := flag[i];
4 while (!b) {
5 b := flag[i];
6 }
7 <wait[i] := false>;
8 r := x; x := r + 1;
9 flag[i] := false;
10 <flag[(i + 1) % TNUM] := true; queue[i] := -1>;

Figure 30. Counter with Anderson array-based lock.

C.2 Counter with Anderson array-based queue lock

In this section, we verify the counter implementation with the Anderson array-based queue lock is linearizable with respect to the atomic operation INC and is starvation-free. Fig. [30] shows the concrete implementation with auxiliary code (in red). Suppose the maximum total number of threads is TNUM. Anderson array-based lock algorithm [13] uses a boolean flag array which contains TNUM slots and a tail pointer which points to the next available slot in the array. To acquire the lock, each thread reads the current value of tail and atomically increments tail (line 2). Then the thread spins until the flag at its slot becomes true (lines 3–6). If flag[i] is true, then the thread with slot i has the permission to acquire the lock. To release the lock, the thread sets the flag at its slot to false (line 9) and then sets the flag at the next slot to true (line 10) to let the next thread acquire the lock.

To verify the code, we introduce an auxiliary array queue which contains TNUM slots (just like the flag array in the original code). The queue array records the thread ID t if the corresponding slot at the flag array is held by the thread t. For an available slot i (i.e., which is not held by any thread), queue[i] is −1. Thus we can know from the queue array the list of threads requesting the lock. Similarly to the previous example of the counter with ticket lock, we also give each slot i a waiting bit wait[i] to indicate whether the thread queue[i] is requesting the lock or has entered the critical section.

Fig. [31] shows the full definitions of the precise invariant, the precondition, the rely and guarantee conditions and the definite action. The definitions are similar the corresponding ones for the counter with ticket lock. The most interesting predicate in the definition of the invariant I is queue(d, n1, n2, sz), which gives the queue d of the threads requesting the lock, and the start and the end slots n1 and n2 in the arrays. The definition of queue(d, n1, n2, sz) contains two cases: the first case says the queue of the threads does not step over the boundary of the arrays (thus n1 ≤ n2), while the second case is the opposite. When n1 = n2, the first case means the queue is empty (no threads are requesting the lock), while the second case means the queue is full (all the TNUM threads are requesting the lock).

Note that as shown in the definition of the guaranteed actions SetFlag and RelLock, we view line 10 as the action of releasing the lock and giving up the slot of the current thread. Line 9 is to reset the slot of the current thread, which is viewed as an action happened when the thread is holding the lock.

Fig. [32] shows the proof outline. The proofs are similar to the proofs of the counter with ticket lock.
\[ l' := e \mid t::l \]

\[ \text{list2set}(e) \overset{\text{def}}{=} \emptyset \]

\[ \text{list2set}(t::l) \overset{\text{def}}{=} \{t\} \cup \text{list2set}(l) \]

\[ I \overset{\text{def}}{=} \exists z, n_1, n_2. \ (\text{TNUM} = z) \ast \text{lock}(n_1, n_2, z) \ast (\text{wait}[n_1] \ast \text{resource} \lor \neg \text{wait}[n_1]) \ast \text{waits}(n_1, n_2, z) \]

\[ \text{resource} \overset{\text{def}}{=} \left(x = \lambda \right) \quad \text{res} \overset{\text{def}}{=} (x = \lambda) \]

\[ \text{lock}(n_1, n_2, z) \overset{\text{def}}{=} \exists d, b. \text{lock}(d, n_1, n_2, z, b) \quad \text{lock}^r_{\text{rl}}(n_1, n_2, z) \overset{\text{def}}{=} \exists d, b. \text{lock}^r_{\text{rl}}(d, n_1, n_2, z, b) \]

\[ \text{lock}(d, n_1, n_2, z, b) \overset{\text{def}}{=} \text{queue}(d, n_1, n_2, z) \ast (\text{tail}[sz = n_2] \ast \text{flags}(n_1, n_2, z) \ast (\text{flag}[n_1] = b) \land (b \lor (\neg b) \land (n_1 < n_2)) \]

\[ \text{lock}^r_{\text{rl}}(d, n_1, n_2, z, b) \overset{\text{def}}{=} \text{lock}(d, n_1, n_2, z, b) \land (t \not\in \text{list2set}(d)) \]

\[ \text{flags}(n_1, z) \overset{\text{def}}{=} (\exists 0 \leq i < n_1 \land n_1 < i < z (\neg \text{flag}[i])) \]

\[ \text{queue}(d, n_1, n_2, z) \overset{\text{def}}{=} \left(0 \leq n_1 \leq n_2 < z \land \text{free}(0, n_1) \ast \text{thrs}(d, n_1, n_2) \ast \text{free}(n_2, z) \lor (0 \leq n_2 \leq n_1 < z) \land \exists d_1, d_2. (d = d_1 :: d_2) \land \text{thrs}(d_2, 0, n_2) \ast \text{free}(n_2, n_1) \ast \text{thrs}(d_1, n_1, z) \right) \]

\[ \text{thrs}(d, n, n') \overset{\text{def}}{=} (d = e) \land (n = n') \land \exists d'. (d = t::d') \land (t \not\in \text{list2set}(d')) \land (\text{queue}[n] = t) \land \text{thrs}(d', n + 1, n') \]

\[ \text{free}(n, n') \overset{\text{def}}{=} (\exists n_0 < n' \ast \text{queue}[i] = -1) \]

\[ \text{waits}(n_1, n_2, z) \overset{\text{def}}{=} \left(0 \leq n_1 \leq n_2 < z \land (\exists 0 \leq i < n_1 \land n_1 < i < z (\neg \text{wait}[i])) \ast (\exists n_1 < i < z \ast \text{wait}[i]) \lor (0 \leq n_2 < n_1 < z) \land (\exists 0 \leq i < n_2 \ast \text{wait}[i]) \ast (\exists n_2 < i < z \ast \text{wait}[i]) \right) \]

\[ P_1 \overset{\text{def}}{=} \exists z, n_1, n_2. (\text{TNUM} = z) \ast \text{lock}^r_{\text{rl}}(n_1, n_2, z) \ast (\text{wait}[n_1] \ast \text{resource} \lor \neg \text{wait}[n_1]) \ast \text{waits}(n_1, n_2, z) \land (1 \leq t \leq z) \]

\[ R_\text{tl} \overset{\text{def}}{=} \bigvee_{t' \neq t} G_{t'} \]

\[ G_t \overset{\text{def}}{=} \exists z. (\text{ReqLock}_z(s) \lor \text{AcqResource}_z(s) \lor \text{SetFlag}_z(s) \lor \text{RelLock}(s) \lor \text{Id}) \ast ([\text{TNUM} = z] \land [1 \leq t \leq z]) \land \text{Id} \land (I \iff I) \]

\[ \text{ReqLock}_z(s) \overset{\text{def}}{=} \exists d, n_1, n_2, b. ((\text{lock}^r_{\text{rl}}(d, n_1, n_2, z, b) \ast (\neg \text{wait}[n_2])) \land (\text{lock}(d::d_1, (n_2 + 1)\%sz, sz, b) \ast \text{wait}[n_2])) \]

\[ \text{AcqResource}_z(s) \overset{\text{def}}{=} \exists d, n_1, n_2. (\text{lock}(d::d, n_1, n_2, sz, true) \ast \text{wait}[n_1] \ast \text{resource}) \land (\text{lock}(d::d, n_1, n_2, sz, true) \ast (\neg \text{wait}[n_1])) \]

\[ \text{SetFlag}_z(s) \overset{\text{def}}{=} \exists d, n_1, n_2. \text{lock}(t::d, n_1, n_2, sz, true) \land \text{lock}(t::d, n_1, n_2, sz, false) \]

\[ \text{RelLock}_z(s) \overset{\text{def}}{=} \exists d, n_1, n_2. \text{lock}(t::d, n_1, n_2, sz, true) \land \text{lock}(t::d, (n_1 + 1)\%sz, n_2, sz, true) \ast \text{resource} \]

\[ D_t \overset{\text{def}}{=} \forall n_1. \text{dp}_n(n_1) \sim \text{dp}_r(n_1) \]

\[ \text{dp}_n(n_1) \overset{\text{def}}{=} \exists d, n_2, sz, b. \text{lock}(t::d, n_1, n_2, sz, b) \ast \text{true} \land I \]

\[ \text{dp}_r(n_1) \overset{\text{def}}{=} \exists d, n_2, sz. \text{lock}^r_{\text{rl}}(d, (n_1 + 1)\%sz, n_2, sz, true) \ast \text{true} \land I \]

**Figure 31.** Invariant, precondition, rely/guarantee and definite action of counter with Anderson array-based lock.
tlocked((tl1, tl2, n1, n2, sz, b) ≜
tqueue((tl1, tl2, n1, n2, sz) * (tail % sz = n2) * fflags(n1, sz) * (flag[n1] = b)
∧ (list2set(tl1) ∩ list2set(tl2) = ∅) ∧ (b ∨ ¬(b) ∧ (n1 < n))

tqueue((tl1, tl2, n1, n2, sz) ≜
(0 ≤ n1 ≤ n < n2 < sz) ∧ free(0, n1) * thrs(tl1, n1, n) * thrs(ttl2, n, n2) * free(n2, sz)
∨ (0 ≤ n2 ≤ n1 ≤ n < sz) ∧ thrs(ttl1, n1, n) * thrs(ttl2, n, n2) * free(n2, sz)
∧ thrs(ttl2, 0, n2) * free(n2, n1) * thrs(ttl1, n1, n) * thrs(ttl2, n, n2)
∧ thrs(ttl2, 0, n) * thrs(ttl1, n1, n) * free(n2, n1) * thrs(ttl1, n1, n, sz)

P0(n1, n2, b) ≜ ∃tl1. Ptl1(n1, n1, n, n2, b) P5(n1, n2, b) ≜ ∃tl1. Ptl1(ttl1, n1, n, n2, b)
P1(n1, n2, b) ≜ ∃tl2, sz. (TNUM = sz) * tlocked((tl1, tl2, n1, n, n2, sz, b)
∧ (wait[n1] * resource ∨ ¬wait[n1]) ∧ (n1 ≠ n)) * waits(n1, n2, sz) ∧ (1 ≤ t ≤ sz)
P2(n1, n2, b) ≜ ∃sz, tl. (TNUM = sz) * lock(ttl, n1, n2, sz, b) * wait[n1] * resource * waits(n1, n2, sz) ∧ (1 ≤ t ≤ sz)

bef(b) ≜ if b = true bef(b) ≜ 1 if b = false

inc():
1  local i, b, r;
   { P ∧ arem(INC) }
2   < i := getAndInc(tail) % TNUM; queue[i] := cid; wait[i] := true; >
   { 3n1, n, n2, b. P0(n1, n, n2, b) ∧ (i = n) ∧ arem(INC) }
3   b := flag[i];
   { 3n1, n2, b. P1(n1, n2, true) ∧ (i = n1) ∧ arem(INC) }
4     <wait[i] := false;{ resource = 3n1, n2, P2(n1, n2, true) ∧ (i = n1) ∧ arem(INC) }
5       r := x; x := r + 1;
   { x = x + 1 }< ∃n1, n2, P2(n1, n2, true) ∧ (i = n1) ∧ arem(INC) }
6     flag[i] := false;
   { (x = X + 1) * (3n1, n2, P2(n1, n2, false) ∧ (i = n1)) ∧ arem(INC) }
7   <flag[i + 1] % TNUM := true; queue[i] := ¬1;{ P ∧ arem(skip) }

Here the loop at lines 4–6 is verified in Figure 33.

Figure 32. Proof outline.
We can prove:

(1) $J \Rightarrow I; Q \Rightarrow I; \text{Sta}(J, G' \land R)$.

(2) $J \Rightarrow (R, G': D \vdash Q)$ (where picking $D'$ used in the WHL rule as $D$).

\textbf{Figure 33.} Proof outline – the loop at lines 4–6.
We also have an additional node. The auxiliary variable plays a similar role as the auxiliary array in the previous proofs of Anderson array-based lock. Since the list is virtual, we introduce two mappings to its predecessor's node and its current-owned node's succwait field to 0. That is, when releasing the lock, the current thread will reuse the predecessor’s node and its original node is no longer owned by any thread. Line 11 physically sets the thread-local pointer mynode, but actually at line 10 of releasing the lock, the thread already switches to own the predecessor node.

Fig. 35 shows the full definitions of the precise invariant, the precondition, the rely and guarantee conditions and the definite action. The definitions are similar to the corresponding ones for Anderson array-based lock. Since the list is virtual, we introduce two mappings ta and tb to record the address and the boolean succwait field of each thread's node. Then as shown in the definition of queue(tl, x, ta, tb) (which is the key to define the invariant $I$), the thread queue is recorded in tq and the nodes of these threads constitute a virtual list seg(tl, x, z, ta, tb). We also have an additional node node($x$, false, $0$) as the predecessor of the virtual list. If the virtual list is empty, then ta points to this additional node $x$; otherwise tail points to the end node of the virtual list. The first thread $t$ at the virtual list acquires the lock. Its succwait field must be true, indicating that its successor thread must be waiting now. Its predecessor $x$’s succwait field is false, saying that $t$ is no longer waiting and has acquired the lock.

Fig. 36 shows the proof outline, which is similar to the previous proofs of Anderson array-based lock.

---

initialize(){
    tail := new Node(false, 0);
    tq := ϵ; acquired := false;
}

thread_initialize(){ // mynode: each thread’s local variable (pointing to a shared node)
    mynode := new Node(false, cid); // fields: succwait, tid
}

inc(){
    local p, b, r;
    mynode.succwait := true;
    <p := getAndSet(&tail, mynode); tq := tq::cid>;
    b := p.succwait;
    while (b) {
        b := p.succwait;
        <acquired := true>;
        r := x; x := r + 1;
        <mynode.succwait := false; tq := getTail(tq); acquired := false; tq::cid; mynode.tid := 0>;
        mynode := p;
    }
}

Figure 34. Counter with CLH lock.

C.3 Counter with CLH list-based queue lock

In this section, we verify the counter implementation with the CLH list-based queue lock is linearizable with respect to the atomic operation INC and is starvation-free. Fig. 34 shows the concrete implementation with auxiliary code (in red). The CLH lock algorithm uses a virtual linked list, with the tail pointer pointing to the end of the list. Each list node contains a field succwait, which is a per-thread flag. It is set to true when the lock is acquired by some thread and reset to false when the lock is released. We view the list as “virtual” because there is no explicit “next” pointer in each node pointing to the successor node, and each thread refers to its predecessor through a thread-local pointer $p$ in the code.

Initially the list contains only the tail node, and each thread has its own node mynode (here mynode is a thread-local pointer, but the node is shared, since the successor thread may keep a reference to the node). To acquire the lock, the thread first sets its succwait field to true to indicate that the thread is requesting the lock (line 2). Next it makes its own node the tail of the list, simultaneously getting a reference $p$ to its predecessor, i.e., the original tail node (line 3). Then the thread spins on the predecessor’s succwait field until the predecessor releases the lock (lines 4–7). To release the lock, the thread sets its succwait field to false (line 10). It reuses its predecessor’s node $p$ as its new own node for future lock accesses (line 11).

To verify the code, we give each list node an auxiliary field $tid$ to record the thread which owns the node. If no thread owns the node, the auxiliary $tid$ field is 0. We also introduce an auxiliary variable $tq$ which is a mathematical list with "::" for concatenation. $tq$ plays a similar role as the auxiliary array queue in the previous proofs of Anderson array-based lock. It records the queue of the threads requesting the lock. The auxiliary variable acquired plays a similar role as the auxiliary array $wait$ in the previous proofs of Anderson array-based lock, but acquired is not a per-thread flag. It is set to true when the lock is acquired by some thread and reset to false when the lock is released.

In the code, we append the current thread ID $cid$ at the end of the thread queue $tq$ at the same time when the thread makes its own node the tail of the list (line 3). The auxiliary variable acquired is set to true after the spinning loop (line 8). We view line 10 as the time of releasing the lock, so we remove the head thread in $tq$ and reset acquired at the same time. Simultaneously we set the predecessor node $p$’s tid field to $cid$ and the currently-owned node’s tid field to 0. That is, when releasing the lock, the current thread will reuse the predecessor’s node and its original node is no longer owned by any thread. Line 11 physically sets the thread-local pointer mynode, but actually at line 10 of releasing the lock, the thread already switches to own the predecessor node.

Fig. 35 shows the full definitions of the precise invariant, the precondition, the rely and guarantee conditions and the definite action. The definitions are similar to the corresponding ones for Anderson array-based lock. Since the list is virtual, we introduce two mappings $ta$ and $tb$ to record the address and the boolean succwait field of each thread’s node. Then as shown in the definition of queue($tl$, $x$, $ta$, $tb$) (which is the key to define the invariant $I$), the thread queue is recorded in $tq$ and the nodes of these threads constitute a virtual list seg($tl$, $x$, $z$, $ta$, $tb$). We also have an additional node node($x$, false, $0$) as the predecessor of the virtual list. If the virtual list is empty, then $ta$ points to this additional node $x$; otherwise $tail$ points to the end node of the virtual list. The first thread $t$ at the virtual list acquires the lock. Its succwait field must be true, indicating that its successor thread must be waiting now. Its predecessor $x$’s succwait field is false, saying that $t$ is no longer waiting and has acquired the lock.

Fig. 36 shows the proof outline, which is similar to the previous proofs of Anderson array-based lock.
tl ::= ε | t::tl  \quad ta ∈ ThrdID → Nat  \quad tb ∈ ThrdID → Bool

\text{list2set}(ε) \overset{def}{=} \emptyset
\text{list2set}(t::tl) \overset{def}{=} \{t\} \cup \text{list2set}(tl)

I \overset{def}{=} \exists sz, tl, x₀, ta, tb. (\text{TNUM} = sz) * \text{lock}(tl, x₀, ta, tb, sz) * ((\neg \text{acquired}) * \text{resource} \lor \text{acquired} \land (tl \neq ε))
\text{resource} \overset{def}{=} (x = X)  \quad \text{tres} \overset{def}{=} (x = ∞)
\text{lock}(tl, x₀, ta, tb, sz) \overset{def}{=} (\text{dom}(ta) = \text{dom}(tb) = \{1, \ldots, sz\})
\land \text{queue}(tl, x₀, ta, tb) * \text{nodoset}(\{t | (1 \leq t \leq sz) \land (t \notin \text{list2set}(tl))\}, ta, tb)
\text{lock}_{\text{irr}}(tl, x₀, ta, tb, sz) \overset{def}{=} (t \notin \text{list2set}(tl)) \land \text{lock}(tl, x₀, ta, tb, sz)
\text{queue}(tl, x₀, ta, tb) \overset{def}{=} \exists sz. (\text{queue}(tl, x₀, ta, tb) * \text{seg}(tl, x₀, ta, tb) * (\text{tail} = z)
\text{seg}(tl, x₀, ta, tb) \overset{def}{=} (tl = ε) \land \text{emp} \land (x = z)
\lor \exists t, t'. (tl = t::t') \land (t \notin \text{list2set}(t')) \land (ta(t) = y) \land (tb(t) = true) \land \text{node}(y, t, true, t) * \text{seg}(t', y, z, ta, tb)
\text{nodoset}(S, ta, tb) \overset{def}{=} (\forall t ∈ S. \text{node}(u(t), tb(t), t))
\text{node}(x, b, s) \overset{def}{=} (x.\text{succwait} = b) * (x.\text{tid} = s)

R₁ \overset{def}{=} R₁(\text{amnode}, false)
R₂ (x, b) \overset{def}{=} \exists x₀, ta, tb, sz, tl. (\text{TNUM} = sz) * \text{lock}_{\text{irr}}(tl, x₀, ta \uplus \{t \sim x\}, tb \uplus \{t \sim b\}, sz)
\land (\neg \text{acquired}) * \text{resource} \lor \text{acquired} \land (tl \neq ε) \land (1 \leq t \leq sz)

R₃ \overset{def}{=} \bigvee_{I \neq I} G₃
G₃ \overset{def}{=} \exists sz. (\text{SetSuccwait} (sz) \lor \text{ReqLock}_1(sz) \lor \text{AcqResource}_1(sz) \lor \text{RelLock}_1(sz) \lor \text{Id})
\land ([\text{TNUM} = sz] \land (1 \leq t \leq sz)) \land (I \neq I)
\text{SetSuccwait} (sz) \overset{def}{=} \exists tl, x₀, ta, tb. \text{lock}_{\text{irr}}(tl, x₀, ta, tb \uplus \{t \sim false\}, sz) \times \text{lock}_{\text{irr}}(tl, x₀, ta, tb \uplus \{t \sim true\}, sz)
\text{ReqLock}_1 (sz) \overset{def}{=} \exists tl, x₀, ta, tb. \text{lock}_{\text{irr}}(tl, x₀, ta, tb, sz) \times \text{lock}(tl::tl, x₀, ta, tb, sz)
\text{AcqResource}_1 (sz) \overset{def}{=} \exists tl, x₀, ta, tb. (\text{lock}(tl::tl, x₀, ta, tb, sz) * (\neg \text{acquired}) * \text{resource}) \times (\text{lock}(tl::tl, x₀, ta, tb, sz) * \text{acquired})
\text{RelLock}_1 (sz) \overset{def}{=} \exists tl, x₀, ta, tb. (\text{lock}(tl::tl, x₀, ta, tb, sz) * \text{acquired})
\times (\text{lock}_{\text{irr}}(tl, ta(t), ta(t \sim x₀), tb(t \sim false), sz) * (\neg \text{acquired}) * \text{resource})
\text{D₃} \overset{def}{=} \forall x₀, ta. dp₃(x₀, ta) \sim dp₃(x₀, ta)
\text{dp₃}(x₀, ta) \overset{def}{=} \exists tl, ta, x₀, ta. (\text{lock}(tl::tl, x₀, ta, tb, sz) * \text{true} \land I)
\text{dp₃}(x₀, ta) \overset{def}{=} \exists tl, ta, x₀, ta. (\text{lock}_{\text{irr}}(tl, ta(t), ta(t \sim x₀), tb(t \sim false), sz) * \text{true} \land I)

\text{Figure 35.} Invariant, precondition, rely/guarantee and definite action.

70
Here the loop at lines 5-7 is verified in Figure 37.

Figure 36. Proof outline.
\[p' \overset{\text{def}}{=} \exists x_0. (P_3(x_0, p, x, \text{true}) \land \Diamond (\text{bet}(b))) \land (p = p) \land (\text{mynode} = x) \land b \land \text{arem(INC)}
\]
\[J \overset{\text{def}}{=} \exists x_0. P_0(x_0, p, x, \text{true})
\]
\[Q \overset{\text{def}}{=} \text{Enabled}(D)
\]
\[G' \overset{\text{def}}{=} [I]
\]
\[f(\emptyset) = k \iff \exists t_1, x_0. (\emptyset \models P_0(t_1, x_0, p, x, \text{true})) \land (k = \text{len}(t_1))
\]
\[\text{len}(\epsilon) \overset{\text{def}}{=} 0 \quad \text{len}(t::t') \overset{\text{def}}{=} 1 + \text{len}(t)
\]
\[(\exists x_0. (p = x) \land (\text{mynode} = x) \land P_0(x_0, p, x, \text{true}) \land b \land \text{arem(INC)} \land \Diamond (\text{bet}(b)) \land Q \land \text{true})
\]
\[\implies (p = p) \land (\text{mynode} = x) \land P_1(p, x, \text{true}) \land b \land \text{arem(INC)} \land \Diamond (0)) \land (\Diamond (1) \land \text{emp})
\]
\[\{p\}
\]
\[\{\exists x_0. (P_3(x_0, p, x, \text{true}) \land \Diamond (0) \lor P_3(x_0, p, x, \text{true}) \land \Diamond (\text{bet}(b))) \land (p = p) \land (\text{mynode} = x) \land b \land \text{arem(INC)}\}
\]
\[6 \quad \text{b := p.succwait;}
\]
\[\{\exists x_0. P_0(x_0, p, x, \text{true}) \land (\neg b) \land (p = x_0) \lor b) \land (p = p) \land (\text{mynode} = x) \land \text{arem(INC)} \land \Diamond (\text{bet}(b))\}
\]

We can prove:

(1) \(J \Rightarrow I; Q \Rightarrow I; \text{Sta}(J, G' \lor R)\).

(2) \(J \Rightarrow \langle R, G'; D \xrightarrow{L} Q\rangle\) (where picking \(D\) needed in the \text{WHL} rule as \(D\)).

\textbf{Figure 37.} Proof outline – the loop at lines 5–7.
initialize()
    tail := null; tq := ϵ; acquired := false;
}

thread_initialize(){ // mynode: each thread's local variable (pointing to a shared node)
    mynode := new Node(false, null, cid); // fields: locked, next, tid
}

inc()
1    local p, s, b, nos, r;
2    <p := getAndSet(&tail, mynode); tq := tq::cid>;
3    if (p != null) {
4        mynode.locked := true;
5        p.next := mynode;
6        b := mynode.locked;
7        while (b) {
8            b := mynode.locked;
9        }
10    }
11    <acquired := true>;
12    r := x; x := r + 1;
13    s := mynode.next;
14    if (s = null) {
15        <nos := cas(&tail, mynode, null); if(nos) { tq := getTail(tq); acquired := false; } >;
16        if (!nos) {
17            s := mynode.next;
18            while (s = null) {
19                s := mynode.next;
20            }
21        }
22    }
23    if (s != null) {
24        <s.locked := false; tq := getTail(tq); acquired := false>; mynode.next := null;
25    }
26}

Figure 38. Counter with MCS lock.

C.4 Counter with MCS list-based queue lock

In this section, we verify the counter implementation with the MCS list-based queue lock is linearizable with respect to the atomic operation INC and is starvation-free. Fig. [38] shows the concrete implementation with auxiliary code (in red). Unlike the CLH lock algorithm, the MCS lock algorithm uses an explicit linked list, with tail pointing to the end of the list. Each list node contains a locked field, indicating whether the corresponding thread is waiting for the lock, and a next field, pointing to the next node in the list.

Initially the list is empty and tail points to null. Each thread owns a node mynode. To acquire the lock, a thread appends its own node at the tail of the list (line 2). If the queue is not empty originally, then the thread sets its predecessor node’s next field to refer to its own node (line 5). The thread then spins on the locked field of its own node, waiting until its predecessor sets this field to false (lines 6–9).

To release the lock, the thread checks whether it has a successor (lines 13–22). It first checks if its node’s next field is null (line 14). If so, then either no other thread is requesting the lock, or there is another thread but it is slow. To distinguish the two cases, the current thread checks if its own node is at the tail of the list, and set tail to null if it is the case (line 15). Otherwise, its own node is not at the tail, thus we know some successor thread is requesting the lock. Then the current thread spins until the successor thread links its node as the next node of the current thread (lines 17–20). When the successor node appears, the current thread sets its successor’s locked field to false (line 24), indicating that the successor has the lock and does not need to wait anymore.

To verify the code, we introduce several auxiliary structures following the proofs of the CLH lock. We give each list node an auxiliary field tid to record the thread which owns the node. Unlike the CLH lock algorithm, here a thread does not change its own node, thus this tid field is set at the beginning and no longer modified during the executions. We also introduce the auxiliary variable tq to record the list of the threads requesting the lock. The auxiliary variable acquired indicates whether the lock has been acquired by some thread.

In the code, we append the current thread ID cid at the end of the thread queue tq at the same time when the thread makes its own node at the tail of the list (line 2). The auxiliary variable acquired is set to true after the spinning loop (line 11). When we are sure that there is no successor thread (i.e., the cas succeeds at line 15), the current thread releases the lock by removing the head thread in tq and resetting acquired at the same time. If there exists some successor thread, the lock is released at line 24.

Fig. [39] defines the precise invariant and the precondition. The lock satisfies either unlocked or locked. For the case of locked, we know the threads requesting the lock constitute the queue tq. The head thread in tq acquires the lock. Its locked bit must be false since it does not need to wait for the lock. Each remaining thread in tq is acquired, so either its locked bit is true and the thread has set its predecessor node’s next field to refer to its own node, or we record the corresponding thread in a set S (the thread is slow and has not linked its node as the next of its predecessor), as defined in the predicate llS. For all the other threads which do not request the lock, we use nodeset for their nodes in
\[
\begin{align*}
\text{tl} & ::= \epsilon \mid t::tl \\
\text{ta} & \in \text{ThrdID} \rightarrow \text{Nat} \\
\text{tb} & \in \text{ThrdID} \rightarrow \text{Bool} \\
\text{list2set}(\epsilon) & \overset{\text{def}}{=} \emptyset \\
\text{list2set}(t::tl) & \overset{\text{def}}{=} \{t\} \cup \text{list2set}(tl)
\end{align*}
\]

\[I \overset{\text{def}}{=} \exists sz, tl, ta, tb, S. (\text{TNUM} = sz) \land \text{lock}(tl, ta, tb, sz, S) \land \text{res}(\text{lock}(tl, ta, tb, sz, S)) \land \text{res}(\text{acquired}(\text{lock}(tl, ta, tb, sz, S))) \land (d \neq \epsilon)
\]

\[\text{release} \overset{\text{def}}{=} (x = X) \quad \text{res} \overset{\text{def}}{=} (x = \_)
\]

\[\text{lock}(tl, ta, tb, sz, S) \overset{\text{def}}{=} \exists y. \text{lock}(tl, ta, tb, y, sz, S)
\]

\[\text{lock}(tl, ta, tb, y, sz, S) \overset{\text{def}}{=} \begin{cases} 
\text{(dom}(ta) = \text{dom}(tb) = \{1, \ldots, sz\}) \\
\land (\text{unlocked}(tl) \land (S = \emptyset) \lor \text{locked}(tl, ta, tb, y, sz, S)) \land \text{nodelist}(\{t | (1 \leq t \leq sz) \land (t \not\in \text{list2set}(d))\}), ta, tb)
\end{cases}
\]

\[\begin{align*}
\text{lock}_\text{irr}(tl, ta, tb, y, sz, S) & \overset{\text{def}}{=} \exists x. (t \not\in \text{list2set}(d)) \land (\text{dom}(ta) = \text{dom}(tb) = \{1, \ldots, sz\}) \land (u(t) = x) \land (b(t) = false) \\
\land (\text{unlocked}(tl) \lor \text{locked}(tl, ta, tb, S) \land \text{nodelist}(x, false, y) \land \text{nodelist}(\{t' | (1 \leq t' \leq sz) \land (t' \neq t) \land (t' \not\in \text{list2set}(d))\}), ta, tb)
\end{align*}
\]

\[\text{unlocked}(tl) \overset{\text{def}}{=} (tq = t) \land (\text{tail} = \text{null}) \land (d = \epsilon) \\
\text{locked}(tl, ta, tb, S) \overset{\text{def}}{=} \exists y. \text{locked}(tl, ta, tb, y, S)
\]

\[\begin{align*}
\text{locked}(tl, ta, tb, y, S) & \overset{\text{def}}{=} \exists t', x, z. (tl = t::t') \land (t \not\in \text{list2set}(d')) \land (u(t) = x) \land (b(t) = false) \\
\land (tq = t) \land \text{nodelist}(x, false, y) \land \text{lls}(t', x, y, z, null, ta, tb, S) \land (\text{tail} = z)
\end{align*}
\]

\[\begin{align*}
\text{lls}(tl, ta, tb, y, S) & \overset{\text{def}}{=} (d = \epsilon) \land (p = z) \land (x = n) \land (S = \emptyset) \\
\land \exists t', x', b, S'. (tl = t::t') \land (t \not\in \text{list2set}(d')) \land (u(t) = x') \land (b(t) = b) \\
\land \text{nodelist}(x', b, y) \land \text{lls}(t', x', y, z, n, ta, tb, S') \land ((x = x') \land (b = true) \land (S = S') \lor (x = null) \land (S = \{t \cup S\}))
\end{align*}
\]

\[\begin{align*}
\text{nodelist}(S, ta, tb) & \overset{\text{def}}{=} \{ \exists s \in S (\text{nodelist}(u(t), false, s) \land (b(t) = false)) \}
\end{align*}
\]

\[\begin{align*}
\text{nodelist}(x, b, y) & \overset{\text{def}}{=} (x \in S \land \text{node}(u(t), false, s) \land (b(t) = false))
\end{align*}
\]

\[\begin{align*}
\text{P}_1 & \overset{\text{def}}{=} \text{P}(\text{mynode}, \text{null}) \\
\text{P}(x, y) & \overset{\text{def}}{=} \exists a, b, sz, tl, S. (\text{TLNUM} = sz) \land \text{lock}_\text{irr}(tl, ta \cup \{t \sim x\}, tb, y, sz, S) \\
\land (\text{acquire}(\text{lock}(tl, ta, tb, sz, S))) \land (d \neq \epsilon) \land (1 \leq t \leq sz)
\end{align*}
\]

Figure 39. Invariant and precondition.

The shared state. Their locked bits are all false. As in the proof of CLH lock, we use two mappings \(ta\) and \(tb\) to record the address and the boolean locked field of each thread's node.

Fig. 34 defines the rely and guarantee conditions and the definite actions. We have several actions since the code is very fine-grained. The action \(\text{RelLock}\) is for line 2, where the thread makes its node the tail of the list. If it has a predecessor, then \(\text{SetLocked}\) sets its locked bit (line 4), and \(\text{SetPred}\) sets its predecessor node's next to its own node (line 5). The head thread in \(\text{tq}\) does \(\text{AcqResource}\) (line 11) to set acquired and transfer the shared resource. To release the lock, we distinguish the case when there is no successor thread (\(\text{RelLock1}\), corresponding to line 15) and the opposite case (\(\text{RelLock2}\), corresponding to line 24). Finally, the thread does \(\text{ResetNext}\) to set the next field of its own node to \(\text{null}\).

Note here we have two definite actions. \(dp1 \sim dq1\) corresponds to the action \(\text{RelLock2}\), which says the thread holding the lock will finally release the lock by setting the successor thread's locked bit to false. The other definite action \(dp2 \sim dq2\) corresponds to \(\text{SetPred}\), saying that the current slow thread which has requested the lock will finally link its node in the list by setting its predecessor node's next field to refer to its own node.

Fig. 41 and Fig. 42 show the proof outline. The proofs are similar to the previous proofs of Anderson array-based lock. Note that the two loops in the code makes use of the two definite actions respectively.
\[
R_t \overset{\text{def}}{=} V_{t' \neq t} G_{t'}
\]

\[
G_{t'} \overset{\text{def}}{=} \\
\exists sz. (\text{ReqLock}_{t'}(sz) \lor \text{SetLocked}_{t'}(sz) \lor \text{SetPred}_{t'}(sz) \lor \text{AcqResource}_{t'}(sz) \lor \text{RelLock1}_{t'}(sz) \lor \text{RelLock2}_{t'}(sz) \lor \text{ResetNext}_{t'}(sz) \lor \text{Id})
\]

\[
\text{ReqLock}_{t'}(sz) \overset{\text{def}}{=} \\
\exists tl, ta, tb, S. \text{lock}_{irr}(tl, ta, tb, \text{null}, sz, S) \land \text{lock}(tl::t, ta, tb, sz, S \uplus \{t\})
\]

\[
\text{SetLocked}_{t'}(sz) \overset{\text{def}}{=} \\
\exists tl_1, tl_2, ta, tb, S. (\{tl \neq t\} \land \text{lock}(tl_1::t::tl_2, ta, tb \uplus \{t \sim \text{false}, sz, S\}) \land \text{lock}(tl_1::t::tl_2, ta, tb \uplus \{t \sim \text{true}, sz, S\})
\]

\[
\text{SetPred}_{t'}(sz) \overset{\text{def}}{=} \\
\exists tl_1, tl_2, ta, tb, S. \text{lock}(tl_1::t::tl_2, ta, tb \uplus \{t \sim \text{true}, sz, S\}) \land \text{lock}(tl_1::t::tl_2, ta, tb \uplus \{t \sim \text{false}, sz, S\})
\]

\[
\text{AcqResource}_{t'}(sz) \overset{\text{def}}{=} \\
\exists tl, ta, tb, S. (\text{lock}(tl::t, ta, tb, sz, S) \land \text{acquired}) \land \text{resource} 
\]

\[
\text{RelLock1}_{t'}(sz) \overset{\text{def}}{=} \\
\exists ta, tb. (\text{lock}(tl, ta, tb, sz, S) \land \text{acquired}) \land (\text{lock}(tl::t, ta, tb, sz, S \uplus \{t\}) \land \text{acquired})
\]

\[
\text{RelLock2}_{t'}(sz) \overset{\text{def}}{=} \\
\exists tl', ta, tb, S. (\text{lock}(tl::t', ta, tb, sz, S) \land \text{acquired} \land \text{resource}) 
\]

\[
\text{ResetNext}_{t'}(sz) \overset{\text{def}}{=} \\
\exists tl, ta, tb, S. \text{lock}_{irr}(tl, ta, tb, sz, S) \land \text{lock}_{irr}(tl, ta, tb, S \uplus \{t\})
\]

\[
\mathcal{D}_t \overset{\text{def}}{=} (\forall tl, ta, tl_1, tl_2. \text{dp}_{l1}(tl', ta, tl) \sim \text{dp}_{l1}(tl', ta, tl) \land (\forall tl_1, ta, tl_2. \text{dp}_{l2}(tl_1, ta, tl_2) \sim \text{dp}_{l2}(tl_1, ta, tl_2)))
\]

\[
\text{dp}_{l1}(tl', ta, tl) \overset{\text{def}}{=} \exists tl', tb, sz, S. \text{lock}(tl::t':tl', ta, tb, sz, S \uplus \{t\}) \land (tl' \not\in S) \land I
\]

\[
\text{dp}_{l1}(tl', ta, tl) \overset{\text{def}}{=} \exists tl'_1, tb, sz, S. \text{lock}(tl_1::t::tl'_1, ta, tb \uplus \{t \sim \text{false}, ta(t'), sz, S \uplus \{t\}) \land (tl' \not\in S) \land I
\]

\[
\text{dp}_{l2}(tl_1, ta, tl_2) \overset{\text{def}}{=} \exists tl'_2, tb, sz, S. \text{lock}(tl_1::t::tl_2::tl'_2, ta, tb \uplus \{t \sim \text{true}, sz, S \uplus \{t\}) \land (tl' \not\in S) \land I
\]

Figure 40. Rely/guarantee and definite action.
tlocked(d1, d2, p, x, ta, tb, sz, S1, S2) ≡
(dom(ta) = dom(tb) = {1, ..., sz}) ∧ (list2set(d1) ∩ list2set(t::d2) = ∅)
∧ twait(d1, d2, p, x, ta, tb, S1, S2) ∧ nodeset({t' | (1 ≤ t' ≤ sz) ∧ (t' ∉ list2set(t::d2))}, ta, tb)

twait(d1, d2, p, x, ta, tb, S1, S2) ≡
∃ t', t', x', y', z. (d1 = t'::d') ∧ (t' ∉ list2set(t')) ∧ (ta(t') = x') ∧ (tb(t') = false)
∧ (tq = t1::t2) ∧ node(t', x', false, y') ∧ lls(t', x', y', p, x, ta, tb, S1) ∧ lls(t::t2, p, x, null, ta, tb, S2) ∧ (tail = z)
P0(p, x, x', b) ≡ ∃ d1, d2, S1, S2. P0(d1, d2, p, x, x', b, S1, S2)
P2(x, b) ≡ ∃ t, y. P2(t, x, b, y)
P0(d1, d2, p, x, x', b, S1, S2) ≡
∃ sz, ta, tb. (TNUM = sz) ∧ tlocked(d1, d2, p, x, ta ⊔ {t → x'}, tb ⊔ {t → b}, sz, S1, S2)
∧ ((¬acquired) ∨ resource ∨ acquired) ∧ (1 ≤ t ≤ sz)
P1(x, b) ≡ ∃ sz, ta, tb, S. (TNUM = sz) ∧ tlocked(d1, d2, p, x, ta ⊔ {t → x'}, tb ⊔ {t → b}, sz, S) ∧ (¬acquired) ∨ resource ∧ (1 ≤ t ≤ sz)
P2(t, x, b, y) ≡ ∃ sz, ta, tb, S. (TNUM = sz) ∧ lock(t::t, ta ⊔ {t → x}, tb ⊔ {t → b}, y, sz, S) ∧ acquired ∧ (1 ≤ t ≤ sz)

bet(b) ≡ 1 if b = true  ∥  0 if b = false
pez(s) ≡ 1 if s = null  ∥  0 if s ≠ null

inc();
    1 local p, s, b, nos, r;
        { P ∧ arem(INC) }
    2 < p := getAndSet(&tail, mynode); tq := tq::cid >;
        { (P0(p, null, mynode, false) ∨ (p = null) ∧ P1(mynode, false)) ∧ arem(INC) }
    3 if (p != null) {
        { P0(p, null, mynode, false) ∧ arem(INC) }
    4 mynode.locked := true;
        { P0(p, null, mynode, true) ∧ arem(INC) }
    5 p.next := mynode;
        { (P0(p, mynode, mynode, true) ∧ P1(mynode, false)) ∧ arem(INC) }
    6 b := mynode.locked;
        { (P0(p, mynode, mynode, true) ∧ b ∨ P1(mynode, false)) ∧ arem(INC) }
        { (p = p) ∧ (mynode = x) ∧ (P0(p, x, x, true) ∧ b ∨ P1(x, false)) ∧ arem(INC) ∧ ⊢(bet(b)) }
    7 while (b) {
    8 b := mynode.locked;
    9 }
    10 }
        { P1(mynode, false) ∧ arem(INC) }
    11 < acquired := true >;
        { resource ∧ P2(mynode, false) ∧ arem(INC) }
    12 ...

Here the loop at lines 7–9 is verified in Figure 42.

Figure 41. Proof outline.
\[
p' \triangleq (P_1(x, false) \land \diamond (0)) \lor P_0(p, x, x, true) \land \diamond (bet(b))) \land (p = p) \land (\text{mynode} = x) \land b \land \text{arem}(INC)
\]
\[
\mathcal{D} \triangleq (\forall t', ta, tl. dp_1(t', ta, tl::t_0) \sim dp_2(t_1, ta, tl::t_0)) \land (\forall t_1, ta, tl_2. dp_2(t_1, ta, tl_2::t_0) \sim dp_2(t_1, ta, tl_2::t_0))
\]
\[
J \triangleq (P_0(p, x, x, true) \lor P_1(x, false)) \land (t_0 = t),\text{ where } t_0 \text{ is a logical variable}
\]
\[
Q \triangleq \text{Enabled}(\mathcal{D})
\]
\[
G' \triangleq \{I\}
\]
\[
f(\mathcal{G}) = k \iff
\exists tl_1, tl_2, S_1, S_2. (\mathcal{G} \models P_0(tl_1, tl_2, p, x, true, S_1, S_2) \land (k = \text{len}(tl_1) + |S_1|)
\land (\mathcal{G} \models P_1(x, false)) \land (k = 0)
\]
\[
\text{len}(e) \triangleq 0 \quad \text{len}(tl::d) \triangleq 1 + \text{len}(d) \quad |\emptyset| \triangleq 0 \quad |S \cup \{x\}| \triangleq |S| + 1
\]

\[
(p = p) \land (\text{mynode} = x) \land (P_0(p, x, x, true) \lor P_1(x, false)) \land b \land \text{arem}(INC) \land \diamond (bet(b)) \land Q \iff
\]
\[
\implies ((p = p) \land (\text{mynode} = x) \land P_1(x, false) \land b \land \text{arem}(INC) \land \diamond (0)) \land (\diamond (1) \land \text{emp})
\]
\[
\{p'\}
\]
\[
\{P_1(x, false) \land \diamond (0) \lor P_0(p, x, x, true) \land \diamond (bet(b))) \land (p = p) \land (\text{mynode} = x) \land b \land \text{arem}(INC)\}
\]
8 \quad b := \text{mynode}.locked;
\[
\{P_0(p, x, x, true) \lor b \land P_1(x, false) \land (p = p) \land (\text{mynode} = x) \land \text{arem}(INC) \land \diamond (bet(b))\}
\]

We can prove:

1. \( J \Rightarrow I ; Q \Rightarrow I ; \text{Sta}(J, G' \Rightarrow R) \).
2. \( J \Rightarrow (R, G'; D \rightarrow Q) \) (where picking \( D' \) needed in the \text{WHL} rule as above).

\underline{Figure 42.} Proof outline – the loop at lines 7–9.

\begin{verbatim}
11 . . .
12 \{ resource + P_2(\text{mynode}, false) \land \text{arem}(INC)\}
13 r := x; x := r + 1;
\{ (x = X + 1) \land P_2(\text{mynode}, false) \land \text{arem}(INC)\}
14 s := \text{mynode}.next;
\{ 3tl, y. (x = X + 1) \land P_2(tl, \text{mynode}, false, y) \land ((s = null) \lor (s = y)) \land \text{arem}(INC)\}
15 if (s = null) {
16 \{ (x = X + 1) \land P_2(\text{mynode}, false) \land (s = null) \land \text{arem}(INC)\}
17 <nos := cas(&tail, \text{mynode}, null); if(nos) \{ tqa := getTail(tqa); acquired := false; \} >;
\{ nos \land P \land (s = null) \land \text{arem}(skip) \lor ((\neg nos) \land 3tl, y. (tl \neq \epsilon) \land (x = X + 1) \land P_2(tl, \text{mynode}, false, y) \land \text{arem}(INC))\}
18 if (!nos) {
19 \{ \exists tl, y. (tl \neq \epsilon) \land (x = X + 1) \land P_2(tl, \text{mynode}, false, y) \land \text{arem}(INC)\}
20 s := \text{mynode}.next;
21 \{ \exists tl, y. (tl \neq \epsilon) \land (x = X + 1) \land P_2(tl, \text{mynode}, false, y) \land ((s = null) \lor (s = y)) \land \text{arem}(INC)\}
22 y := \text{mynode}.next;
\{ 3tl, y. (tl \neq \epsilon) \land (x = X + 1) \land P_2(tl, \text{mynode}, false, y) \land \text{arem}(INC) \land \diamond (\text{pez}(s))\}
23 if (s != null) {
24 \{ 3tl, y. (tl \neq \epsilon) \land (x = X + 1) \land P_2(tl, \text{mynode}, false, s) \land (s \neq null) \land \text{arem}(INC)\}
25 <s.locked := false; tqa := getTail(tqa); acquired := false;>
\{ P(\text{mynode}_t) \land \text{arem}(skip)\}
26 \text{mynode.next} := \text{null};
27 \}
\}
\{ P \land \text{arem}(skip)\}
\end{verbatim}

Here the loop at lines 18–20 is verified in Figure 43.

\underline{Figure 43.} Proof outline (continued).
\( p'' \overset{\text{def}}{=} \exists \mathcal{t}, y. (\mathcal{t} \neq \epsilon) \land (\text{mynode} = x) \land (x = X + 1) \land P_2(\mathcal{t}, x, \text{false}, y) \land (s = \text{null}) \land \text{arem}(\text{INC}) \land (y \neq \text{null} \lor \Diamond(0) \lor y = \text{null} \lor \Diamond(\text{pez}(s))) \)

\( D''_t \overset{\text{def}}{=} (\forall \mathcal{t}_1, \mathcal{t}_2. dp_2(\mathcal{t}_1 :: t_0, \mathcal{t}_2) \sim dq_2(\mathcal{t}_1 :: t_0, \mathcal{t}_2)) \)

\( J''_t \overset{\text{def}}{=} \exists \mathcal{t}, y. (\mathcal{t} \neq \epsilon) \land P_2(\mathcal{t}, x, \text{false}, y) \land (t_0 = t) \), where \( t_0 \) is a logical variable

\( Q''_t \overset{\text{def}}{=} \text{Enabled}(D'_t) \)

\( G'' \overset{\text{def}}{=} [t] \)

\( f'(\mathcal{S}) = k \iff \exists \mathcal{t}, y. (\mathcal{t} \neq \epsilon) \land (\mathcal{S} \vdash P_2(\mathcal{t}, x, \text{false}, y)) \land (k = \text{pez}(y)) \)

\( (\exists \mathcal{t}, y. (\mathcal{t} \neq \epsilon) \land (\text{mynode} = x) \land (x = X + 1) \land P_2(\mathcal{t}, x, \text{false}, y) \land (s = \text{null}) \land \text{arem}(\text{INC}) \land \Diamond(\text{pez}(s)) \land Q'' \land \text{true}) \)

\( \implies (\exists \mathcal{t}, y. (\mathcal{t} \neq \epsilon) \land (\text{mynode} = x) \land (x = X + 1) \land P_2(\mathcal{t}, x, \text{false}, y) \land (y \neq \text{null}) \land (s = \text{null}) \land \text{arem}(\text{INC}) \land \Diamond(0)) \)

\( \overset{\text{true}}{\times} \)

\( \{ p'' \} \)

\( \{ \exists \mathcal{t}, y. (\mathcal{t} \neq \epsilon) \land (\text{mynode} = x) \land (x = X + 1) \land P_2(\mathcal{t}, x, \text{false}, y) \land (s = \text{null}) \land \text{arem}(\text{INC}) \land (y \neq \text{null}) \land \Diamond(0) \land \text{null} \land \Diamond(\text{pez}(s))) \} \)

19 \quad s := \text{mynode.next};

\( \{ \exists \mathcal{t}, y. (\mathcal{t} \neq \epsilon) \land (\text{mynode} = x) \land (x = X + 1) \land P_2(\mathcal{t}, x, \text{false}, y) \land ((s = \text{null}) \lor (s = y)) \land \text{arem}(\text{INC}) \land \Diamond(\text{pez}(s)) \} \)

We can prove:

(1) \( J' \Rightarrow I; \, Q' \Rightarrow I; \, \text{Sta}(J', G' \lor R) \).

(2) \( J' \Rightarrow (R, G': D''_t \rightarrow Q') \) (where picking \( D' \) needed in the WHL rule as above \( D''_t \)).

\( \text{Figure 44.} \) Proof outline (continued) – the loop at lines 18–20.
Abstract operations:

\[
\text{ENQ}(V) \{ \ < Q := Q :: V; > \}
\]
\[
\text{DEQ}() \{
\begin{align*}
\text{local } V; \\
\text{< if } (Q = \epsilon) \{ \\
\text{V := EMPTY;} \\
\} \text{ else } \{ \\
\text{V := head(Q); Q := tail(Q);} \\
\}> \\
\}
\]

\[
\text{return V;}
\]

struct Node {
  int data;
  struct Node *next;
}

struct Lock {
  int lowner, lnext;
  int[] ticket; //initially all -1
}

struct Queue {
  struct Node *Head;
  struct Node *Tail;
  struct Lock *Hlock, Tlock;
}

initialize()
{
  Head := cons(0, null);
  Tail := Head;
  Hlock.lowner := Hlock.lnext := Tlock.lowner := Tlock.lnext := 0;
}

enq(v) {
  1 local x, i, o;
  2 x := cons(v, null);
  3 <i := getAndInc(Tlock.lnext); Tlock.ticket := cid;>
  4 <o := Tlock.lowner;>
  5 while (i <> o) {
  6 <o := Tlock.lowner;>
  7 }
  8 Tail.next := x;
  9 Tail := x;
  10 <Tlock.lowner++;>
}

deq() {
  1 local h, s, v, i, o;
  2 <i := getAndInc(Hlock.lnext); Hlock.ticket := cid;>
  3 <o := Hlock.lowner;>
  4 while (i <> o) {
  5 <o := Hlock.lowner;>
  6 }
  7 h := Head;
  8 s := h.next;
  9 if (s = null) {
  10 <Hlock.lowner++;>
  11 v := EMPTY;
  12 } else {
  13 v := s.data;
  14 Head := s;
  15 <Hlock.lowner++;>
  16 dispose(h);
  17 }
  18 return v;
}

Figure 45. Two-lock queue implementation.

C.5 Two-lock queue with ticket lock

In Figure 45, we show the abstract code of the queue algorithm and the concrete implementation of the two-lock queue with auxiliary code (in red). In the concrete implementation, the two locks are implemented using ticket lock. In the algorithm, the queue is implemented as a singly-linked list with the Head and Tail pointers and a sentinel node pointed to by Head. The enq method inserts a new node at the tail of the queue. The deq method replaces the sentinel node by its next node and returns the value in the new sentinel. The Tail and Head pointers are protected by two locks Tlock and Hlock respectively. At any time at most one enq thread and one deq thread can access the queue. But an enq thread and a deq thread do not need to wait for each other.

Since the locks are implemented using the ticket lock algorithm, we introduce the auxiliary tickets to verify the code (as in the proof for the counter with ticket lock). Note here we do not need the auxiliary wait bits as in the proof for the counter with ticket lock, because now we do not have ownership transfers of resource when the lock is acquired/released.

Fig. 46 defines the precise invariant and the precondition. Fig. 47 defines the rely and guarantee conditions and the definite action. The definite action for the whole object is defined as the conjunction of the definite actions on the two locks: \( D \overset{\text{def}}{=} dT \land dH \), where \( dT \) (or \( dH \)) says that the thread must eventually release the lock if it has acquired Tlock (or Hlock).

Fig. 48 and Fig. 50 show the proof outlines. The proof of the algorithm in our logic combines its linearizability proof [21] and the proofs on the progress properties of the ticket lock implementation, which are similar to the above proofs for the counter object with ticket lock. It also relies on the fact that if a thread is holding a lock, it will not request the other lock. Note that the proofs for the loops of acquiring the locks follow the proofs for the counter with ticket lock (see Sec. C.1).
\[
\begin{align*}
\text{tl} & ::= \epsilon \mid t::\text{tl} \\
\text{list2set}(\epsilon) & \equiv \emptyset \quad \text{list2set}(t::\text{tl}) \equiv \{t\} \cup \text{list2set}(\text{tl}) \\
I & \equiv \exists h, z, t' T. (\text{Head} = h) \ast (\text{Tail} = z) \ast \text{queue}(t' T, h, z) \ast \text{lock} \ast \text{lock} \ast \text{lock} \ast \text{lock} \\
\text{queue}(t' T, h, z) & \equiv \exists v_d, A. (Q = A) \ast (\text{unlag}(h, z, v_d :: A) \lor (\text{lag}(h, z, v_d :: A) \land (t' T \neq \epsilon)) \lor (\text{cross}(h, v_d :: A) \land (t' T \neq \epsilon))) \\
\text{unlag}(h, z, A) & \equiv \exists v', A'. (A = A' :: v) \land \text{ls}(h, A', z) \ast N(z, v, \text{null}) \\
\text{lag}(h, z, x, A) & \equiv \exists v, v', A'. (A = A' :: v :: v') \land \text{ls}(h, A', z) \ast N(2(z, v, x, v', \text{null})) \\
\text{cross}(h, A) & \equiv \exists v. (A = v :: \epsilon) \land N(h, v, \text{null}) \\
\text{ls}(x, A, y) & \equiv (x = y \land A = \epsilon) \lor (x \neq y \land \exists z, v, A'. A = v :: A' \land N(x, v, z) \ast \text{ls}(z, A', y)) \\
N(p, v, y) & \equiv (p.\text{data} = v) \ast (p.\text{next} = y) \quad N(2(p, v, y, v', z) & \equiv N(p, v, y) \ast N(y, v', z) \\
\text{lock}(l) & \equiv \exists l. \text{lock}(l, l) \quad \text{lock}(d, l) \equiv \exists n_1, n_2. \text{lock}(d, l, n_1, n_2) \\
\text{lock}(d, l, n_1, n_2) & \equiv ((\text{l.}\text{lower} = n_1) \ast (\text{l.}\text{next} = n_2) \land (n_1 \leq n_2)) \ast \text{tickets}(l, 0, n_1) \ast \text{tickets}(d, l, n_1, n_2) \ast \text{tickets.\text{new}}(l, n_2) \\
\text{tickets}(l, n_1, n_2) & \equiv \exists l. \text{tickets}(d, l, n_1, n_2) \\
\text{tickets}(d, l, n_1, n_2) & \equiv ((d = \epsilon) \land (n_1 = n_2) \land \text{emp} \lor \exists t'. (d = t::t') \land (l.\text{ticket}_{n_1} = t) \ast \text{tickets}(t', l, n_1 + 1, n_2) \\
\text{tickets.\text{new}}(l, n_2) & \equiv (\forall i \geq n_2. \text{ticket}_i = -1) \\
F_l & \equiv \exists h, z, t' T. (\text{Head} = h) \ast (\text{Tail} = z) \ast \text{queue}(t' T, h, z) \ast \text{lock.\text{irr}} \ast \text{lock.\text{irr}} \ast \text{lock.\text{irr}} \ast \text{lock.\text{irr}} \\
\text{lock.\text{irr}}(l) & \equiv \exists l. \text{lock.\text{irr}}(l, l) \quad \text{lock.\text{irr}}(d, l) \equiv \exists n_1, n_2. \text{lock.\text{irr}}(d, l, n_1, n_2) \\
\text{lock.\text{irr}}(d, l, n_1, n_2) & \equiv \text{lock}(d, l, n_1, n_2) \land (t \notin \text{list2set}(d)) \\
\end{align*}
\]
\[ R_t \overset{\text{def}}{=} \bigvee_{i' \neq i} G_{i'} \]

\[ G_t \overset{\text{def}}{=} (\text{Enq} \lor \text{Swap} \lor \text{Deq} \lor \text{ReqLock} \lor \text{Unlock} \lor \text{ReqLock} \lor \text{Unlock} \lor \text{Id}) \land (I \times I) \]

\[ \text{Enq} \overset{\text{def}}{=} \exists t, x, y, A, v, v'. [\text{lock}(t::tl, Tlock) \times (\text{Tail} = x)] \]

\[ \exists (N(x, v, \text{null}) \times (Q = A)) \times (N2(x, v, v', \text{null}) \times (Q = A')) \]

\[ \text{Swap} \overset{\text{def}}{=} \exists t, x, v. [\text{lock}(t::tl, Tlock) \times N(x, v, \text{null})] \times ((\text{Tail} = x) \times (\text{Tail} = x)) \]

\[ \text{Deq} \overset{\text{def}}{=} \exists t, x, y, z, v, v'. A. [\text{lock}(t::tl, Hlock)] \]

\[ \exists (\text{Head} = x) \times N2(x, v, v', z) \times (Q = v'::A) \times ((\text{Head} = y) \times N(y, v', z) \times (Q = A)) \]

\[ \text{ReqLock} \overset{\text{def}}{=} \exists t, n_1, n_2. [\text{lock}(t::tl, Hlock)] \times (\text{lock}(\text{tl}, Hlock, n_1, n_2) \times \text{lock}(\text{tl}, Hlock, n_1, n_2 + 1)) \]

\[ \text{Unlock} \overset{\text{def}}{=} \exists t, n_1, n_2. [\text{lock}(t::tl, Hlock)] \times (\text{lock}(\text{tl}, Hlock, n_1, n_2) \times \text{lock}(\text{tl}, Hlock, n_1 + 1, n_2)) \]

\[ \text{ReqLock} \overset{\text{def}}{=} \exists t, n_1, n_2. [\text{lock}(t::tl, Tlock)] \times (\text{lock}(\text{tl}, Tlock, n_1, n_2) \times \text{lock}(\text{tl}, Tlock, n_1, n_2 + 1)) \]

\[ \text{Unlock} \overset{\text{def}}{=} \exists t, n_1, n_2. [\text{lock}(t::tl, Tlock)] \times (\text{lock}(\text{tl}, Tlock, n_1, n_2) \times \text{lock}(\text{tl}, Tlock, n_1 + 1, n_2)) \]

\[ D_t \overset{\text{def}}{=} (\forall n_1. \quad dpH_t(n_1) \sim dqH_t(n_1)) \land (\forall n_1. \quad dpT_t(n_1) \sim dqT_t(n_1)) \]

\[ dpH_t(n_1) \overset{\text{def}}{=} \exists t, n_2. \text{lock}(t::tl, Hlock, n_1, n_2) \times \text{true} \land I \]

\[ dqH_t(n_1) \overset{\text{def}}{=} \exists t, n_2. \text{lock}(t::tl, Hlock, n_1 + 1, n_2) \times \text{true} \land I \]

\[ dpT_t(n_1) \overset{\text{def}}{=} \exists t, n_2. \text{lock}(t::tl, Tlock, n_1, n_2) \times \text{true} \land I \]

\[ dqT_t(n_1) \overset{\text{def}}{=} \exists t, n_2. \text{lock}(t::tl, Tlock, n_1 + 1, n_2) \times \text{true} \land I \]

---

**Figure 47.** Rely, guarantee and definite actions of the two-lock queue.
\[
ticked_{d_{1},d_{2}}(l,n_{1},n_{2}) \equiv \\
(\text{listset}(d_{1}) \cap \text{listset}(d_{2})) = \emptyset \land ((l.\text{houn} = n_{1}) \land (l.\text{next} = n_{2}) \land (n_{1} \leq n < n_{2})) \\
* \text{tickets}(l,0,n_{1}) \land \text{tickets}(d_{1},l,n_{1},n) \land \text{tickets}(d_{2},l,n_{2}) \land \text{tickets}_{new}(l,n_{2})
\]

**Proof outline for enq.**

\[
P_{0}(n_{1},n_{2}) \equiv \exists d_{1}. \quad P_{0}(d_{1},n_{1},n_{2})
\]

\[
P_{3}(n_{1},n_{2}) \equiv \exists \delta'. \quad d_{1}. \quad P_{0}(\delta',d_{1},n_{1},n_{2})
\]

\[
P_{0}(d_{1},n_{2}) \equiv \\
\exists h,z. \quad \text{Head} = h \land \text{Tail} = z \\
* \text{queue}(d_{1},h,z) \land \text{lock}_{\text{irr}}(\text{Hlock}) \land \text{ticked}_{d_{1},d_{2}}(\text{Lock},n_{1},n_{2})
\]

\[
P_{1}(n_{1},n_{2}) \equiv \\
\exists h,z,v_{d}. \quad \text{Head} = h \land \text{Tail} = z \\
* (Q = A) \land \text{unlock}(h,z,v_{d},A) \land \text{lock}_{\text{irr}}(\text{Hlock}) \land \text{lock}(t::d_{1},\text{Lock},n_{1},n_{2})
\]

\[
P_{2}(x,n_{1},n_{2}) \equiv \\
\exists h,z,v_{d}. \quad \text{Head} = h \land \text{Tail} = z \\
* (Q = A) \land (\text{lag}(h,z,v_{d},A) \lor (\text{cross}(h,v_{d},A) \land (h = x)))
\]

\[
\text{enq}(v) \}\{ \\
1 \quad \text{local} \ x, \ i, \ o; \\
\quad \{ P \land \text{arem(ENQ)} \land (v = V) \} \\
2 \quad x := \text{cons}(v, \text{null}); \\
\quad \{ P \land \text{arem(ENQ)} \land (v = V) \} \\
3 \quad \langle i, o \rangle := \text{getAndInc(\text{Lock},\text{next})}; \quad \text{Lock}.\text{ticket} := \langle \text{cid} \rangle; \\
\quad \{ \exists n_{1},n_{2}. \quad P_{0}(n_{1},n_{2}) \land \text{N}(x,v,\text{null}) \land (i = n) \land \text{arem(ENQ)} \land (v = V) \} \\
4 \quad \langle o \rangle := \text{Lock}.\text{lower}; \\
\quad \{ \exists n_{1},n_{2}. \quad P_{0}(n_{1},n_{2}) \land \text{N}(x,v,\text{null}) \land (i = n) \land \text{arem(ENQ)} \land (v = V) \} \\
\quad \{ \exists n_{1},n_{2}. \quad P_{0}(n_{1},n_{2}) \land \text{N}(x,v,\text{null}) \land (i = n) \land \text{arem(ENQ)} \land (v = V) \land o = n - o \} \\
5 \quad \text{while} (i \neq o) \{ \\
\quad \{ \exists n_{1},n_{2}. \quad P_{1}(n_{1},n_{2}) \land \text{arem(ENQ)} \land (v = V) \} \\
6 \quad \langle o \rangle := \text{Lock}.\text{lower}; \\
7 \quad \{ \exists n_{1},n_{2}. \quad P_{2}(n_{1},n_{2}) \land \text{arem(skip)} \} \\
8 \quad \text{Tail.next} := x; \\
\quad \{ \exists n_{1},n_{2}. \quad P_{3}(x,n_{1},n_{2}) \land \text{arem(skip)} \} \\
9 \quad \text{Tail} := x; \\
\quad \{ \exists n_{1},n_{2}. \quad P_{4}(n_{1},n_{2}) \land \text{arem(skip)} \} \\
10 \quad \langle \text{Lock}.\text{lower}++ \rangle; \\
\quad \{ P \land \text{arem(skip)} \} \\
\}
\]

Here the loop at lines 5-7 is verified in Figure[10]

**Figure 48.** Proof outline for enq.

\[
D_{\gamma} \equiv (\forall n_{1}. \quad \text{dpT}(n_{1}) \rightarrow \text{dpT}(n_{1}))
\]

\[
J \equiv \exists n_{1}. \quad \text{P}_{0}(n_{1},n_{2})
\]

\[
Q \equiv \text{Enabled}(D)
\]

\[
G' \equiv [I]
\]

\[
f(\emptyset) = k \text{ if } \emptyset \models (n - \text{Lock}.\text{houn} = k)
\]

\[
(\exists n_{1},n_{2}. \quad P_{0}(n_{1},n_{2}) \land \text{N}(x,v,\text{null}) \land (i = n) \land o \leq n \leq n + 1 \land (o \neq 1) \land \text{arem(ENQ)} \land (v = V) \land o = n - o \land Q \land true) \\
\implies (\exists n_{1}. \quad \text{P}_{3}(n_{1},n_{2}) \land \text{N}(x,v,\text{null}) \land (i = n) \land o < n \land \text{arem(ENQ)} \land (v = V) \land (o = n + 1)) \land (\text{emp} \land (I)) \\
\]

\[
(\exists n_{1},n_{2}. \quad P_{0}(n_{1},n_{2}) \land \text{N}(x,v,\text{null}) \land (i = n) \land o \leq n \leq n + 1 \land (o \neq 1) \land \text{arem(ENQ)} \land (v = V) \land (o = n - o)) \land (\text{emp} \land (I))
\]

We can prove:

1. \(J \Rightarrow I; \quad Q \Rightarrow I. \quad \text{Sta}(J,G' \lor R)\).

2. \(J \Rightarrow (R,G' ; D' \downarrow Q)\) (where picking \(D'\) needed in the W.HL rule as above).

**Figure 49.** Proof outline for enq – the loop at lines 5-7.
\[ P_0'(n_1, n, n_2) \overset{\text{def}}{=} \exists h_1. P_0'(d_1, n_1, n, n_2) \quad P_0''(n_1, n, n_2) \overset{\text{def}}{=} \exists t', h_1. P_0'(t', d_1, n_1, n, n_2) \]

\[ P_0'(d_1, n_1, n, n_2) \overset{\text{def}}{=} \exists h, z, n_2, d_T. (\text{Head} = h) * (\text{Tail} = z) * \text{queue}(d_T, h, z) * \text{tlocked}(d_1, t, d_2)(Hlock, n_1, n, n_2) * \text{lock}_{\text{irr}}(d_T, Tlock) \]

\[ P_1'(h) \overset{\text{def}}{=} \exists n_1, n_2. P_1'(h, n_1, n_2) \quad P_0'(h, s) \overset{\text{def}}{=} \exists v. P_0'(h, s, v) \]

\[ P_1'(h, n_1, n_2) \overset{\text{def}}{=} \exists x, s, v, x. A \land P_1'(h, s, v) \]

\[ P_0'(h, s, v) \overset{\text{def}}{=} \exists x, s, v, x. A \land P_0'(h, s, v) \]

\[ \exists h, s, v, i, o. (P \land \text{arem}(DEQ)) \]

```java
int deq() {
    1 local h, s, v, i, o;
    \{ P \land \text{arem}(DEQ) \}
    2 <i := getAndInc(Hlock.next); Hlock.ticket_1 := cid>; \{ \exists n_1, n_2. P_0'(n_1, n, n_2) \land (i = n) \land \text{arem}(DEQ) \}
    3 \{ \exists n_1, n_2. P_0'(n_1, n, n_2) \land (i = n) \land \text{arem}(DEQ) \}
    4 while (i < o) { \{ P \land \text{arem}(DEQ) \}
    5 <o := Hlock.lowner++; \{ \exists n_1, n_2. P_0'(n_1, n, n_2) \land (i = n) \land \text{arem}(DEQ) \land (o - n) \}
    6 } \{ P_0'(h) \land \text{arem}(DEQ) \}
    7 h := Head;
    \{ P_0'(h) \land \text{arem}(DEQ) \}
    8 s := h.next;
    \{ ((s = null) \land P_0'(h) \land \text{arem}(skip) \land (V = EMPTY)) \lor (P_0'(h, s) \land \text{arem}(DEQ)) \}
    9 if (s = null) { \{ P_0'(h) \land \text{arem}(skip) \land (V = EMPTY) \}
    10 <Hlock.lowner++; \{ P \land \text{arem}(skip) \land (V = EMPTY) \}
    11 v := EMPTY;
    12 } else { \{ P_0'(h, s) \land \text{arem}(DEQ) \}
    13 v := s.data;
    \{ P_0'(h, s, v) \land \text{arem}(DEQ) \}
    14 Head := s;
    \{ P_0'(s) \land \text{arem}(skip) \land (v = V) \}
    15 <Hlock.lowner++; \{ P \land \text{arem}(skip) \land (v = V) \}
    16 dispose(h);
    17 } \{ P \land \text{arem}(skip) \land (v = V) \}
    18 return v;
}
```

Here the loop at lines 4–6 is verified in Figure 51.
\( p'' \) def \( \exists h, n_1, n_2. (P'_1(h, n, n_2) \land \Diamond(n - (o + 1)) \lor P'_5(n_1, n, n_2) \land \Diamond(n - o)) \)
\( \land (o \leq n_1 \leq n = 1) \land (1 \neq o) \land \text{arem(DEQ)} \)

\( D'_t \) def \( (\forall n_1. dpH_t(n_1) \sim dqH_t(n_1)) \)

\( J' \) def \( \exists n_1, n_2. P'_0(n_1, n, n_2) \)

\( Q' \) def \( \text{Enabled}(D) \)

\( G' \) def \( [I] \)

\( f'(\emptyset) = k \) iff \( \emptyset \models (n - \text{Hlock.lowner} = k) \)

\( (\exists n_1, n_2. P'_0(n_1, n, n_2) \land (i = n) \land (o \leq n_1) \land (i \neq o) \land \text{arem(DEQ)} \land \Diamond(n - o) \land Q' \ast true) \)
\( \iff (\exists h, n_2. P'_1(h, n, n_2) \land (i = n) \land (o < n) \land \text{arem(DEQ)} \land \Diamond(n - o + 1)) \ast ((1) \land \text{emp}) \)

\( \{ p'' \} \\
\{ \exists h, n_1, n_2. (P'_1(h, n_2) \land \Diamond(n - (o + 1)) \lor P'_5(n_1, n, n_2) \land \Diamond(n - o)) \}
\{ \land (o \leq n_1 \leq n = 1) \land (1 \neq o) \land \text{arem(DEQ)} \}
\}

\( \emptyset : \oplus \text{Hlock.lowner} \}
\{ \exists n_1, n_2. P'_0(n_1, n, n_2) \land (i = n) \land (o \leq n_1) \land \text{arem(DEQ)} \land \Diamond(n - o) \}

We can prove:
1. \( J' \Rightarrow I; Q' \Rightarrow I; \text{Sta}(J', G' \lor R) \).
2. \( J' \Rightarrow (R, G'; D' \longrightarrow Q') \) (where picking \( D' \) needed in the \text{WHL} rule as above \( D'' \)).

\textbf{Figure 51.} Proof outline for deq – the loop at lines 4–6.
C.6 Lock-coupling list with ticket lock

In Figure 52 we show the abstract code of the set algorithm and the concrete implementation of the lock-coupling list with auxiliary code (in red and purple). In the concrete implementation, the node locks are all implemented using ticket lock. The algorithm implements an abstract set with \( \text{add} \) and \( \text{rmv} \) operations. The concrete list is an ordered singly-linked list with the head pointer and two sentinel nodes at the two ends of the list containing the smallest and the biggest values respectively. Each list node is associated with a lock. Traversing the list uses hand-over-hand locking; the lock on one node is not released until its successor is locked. \( \text{add}(e) \) inserts a new node with value \( e \) in the appropriate position while holding the lock of its predecessor. \( \text{rmv}(e) \) redirects the predecessor’s pointer when both the node to be removed and its predecessor are locked.

To verify the code, we need some auxiliary structures. Since the locks are implemented using the ticket lock algorithm, we introduce the auxiliary tickets for the lock of each node (as in the previous proof for the two-lock queue with ticket locks). Also, we introduce the auxiliary \( \text{toadd} \) bit for each thread to indicate whether the thread attempts to do the \( \text{add} \) operation. \( \text{toadd} \) is true if the thread \( t \) is calling \( \text{add} \) but has not really inserted the new node into the list. It is false if the thread \( t \) is not calling \( \text{add} \) or it has already inserted the new node. The auxiliary tickets and \( \text{toadd} \) bits are all shared. In addition, we introduce an auxiliary local variable \( \text{aux} \) to encode the metric for the loop of list traversal. Fig. 53 shows the auxiliary code, \( \text{metric}(p) \), to compute the metric for the loop of list traversal. Here \( p \) points to the “predecessor” node during the list traversal. Informally, \( \text{metric}(p) \) is the sum of the current maximal length of the remaining nodes to traverse (i.e., the number of the nodes from \( p \) to the end of the list) and the number of the environment threads accessing the remaining nodes which attempt to do \( \text{add} \) (i.e., these threads are going to add nodes, increasing the maximal length of the nodes remained for the current thread to traverse).

Fig. 54 defines the precise invariant and the precondition. The shared state contains the \( \text{toadd} \) bit for each thread and the list. Each list node contains a ticket lock. The predicate \( \text{ss} \) requires the concrete list should be sorted and its elements should constitute the abstract set.

Fig. 55 defines the rely and guarantee conditions and the definite action. In addition to the actions \( \text{add} \) and \( \text{rmv} \) which successfully inserts and removes a node, the thread may do the \( \text{ToAdd} \) action to set its \( \text{toadd} \) bit to true when it just starts its \( \text{add} \) operation. If the \( \text{add} \) fails (i.e., for the case when the value it attempts to insert has already in the list), the thread does the \( \text{FailedAdd} \) action that simply resets its \( \text{toadd} \) bit to false. The \( \text{ReqLock1} \) action requests the lock of the head node in the list. At that time, the thread must just starts its operation, thus it has not requested the lock of a list node. The \( \text{ReqLock2} \) action requests the lock of a node when the thread has already acquired the lock of the predecessor node. The \( \text{Unlock} \) action releases the lock of the predecessor node.

The difficulty of verifying the lock-coupling list algorithm lies in defining the definite action \( D \). Unlike the counter objects and the two-lock queues, in lock-coupling lists, the thread \( t \) holding a lock does not guarantee to release the lock because \( t \) may first try to acquire the lock of the successor node. In fact, the progress of \( t \) relies on the progress of the environment threads which start their list traversals earlier than the thread \( t \). The nodes being visited by these environment threads might be visited by \( t \) in the future. Therefore we define the definite action \( D \) to describe the progress of the thread which is the first to start the list traversal, i.e., the thread accessing the node closest to the tail of the list. Informally, \( D \) says that \( t \) eventually releases the lock of the node \( x \), if \( t \) holds the lock and all the nodes remained to visit (the nodes from the successor of \( x \) to the end of the list) are either unlocked or locked by \( t \) itself.

The whole proof follows the basic linearizability proof of lock-coupling list (e.g., see [21]). Now we also verify the starvation-freedom of the object.

We first define the general well-founded order and the metrics for the loops in Fig. 56. We show the proof for the \( \text{add} \) operation in Fig. 57 and the proof for the \( \text{rmv} \) operation in Fig. 61. Note we define the \( Q \) needed for verifying the loops similarly as the case when \( D \) is enabled. Actually the outer loop of list traversal always terminates no matter whether \( D \) is enabled or not, because the number of the nodes remained to traverse (including the nodes possibly added in the future by the environment threads) must be decreasing, i.e., the metric function \( \text{metric}(p) \) defined in Fig. 53 must be decreasing at each loop iteration. For the inner loop of requesting the lock of a node \( e \), if \( D \) is enabled, we know there is no thread accessing the node \( e \), then the current thread must succeeds in acquiring the lock and the loop could terminate.

Below we explain how we define \( f \) needed for verifying the loops. As shown in Fig. 56 for any state during the execution of a loop, the metric contains two parts: \( \text{toadd} \), the \( \text{toadd} \) bits of all the threads, and \( L \), the list of the environment threads ahead of the current thread (including the threads which are ahead of the current thread and are requesting the lock of the same node). The well-founded order defined at the top of Fig. 56 says, the metric decreases if (1) the number of the environment threads in \( L \) decreases; or (2) the number of the environment threads which attempt to do \( \text{add} \) and are ahead of the current thread decreases; or (3) some environment thread ahead of the current thread gets closer to the end of the list. Thus the metric must decrease when a relevant environment thread progresses, and the key requirements \( J \Rightarrow (R, G; D \rightarrow Q) \) for verifying loops hold.
(a) Abstract operations.

```c
ADD(E) {
    local R;
    if (E ∈ S) {
        R := false;
    } else {
        S := S ∪ {E}; R := true;
    }
    return R;
}

RMV(E) {
    local R;
    if (E ∉ S) {
        R := false;
    } else {
        S := S \ {E}; R := true;
    }
    return R;
}
```

(b) Concrete implementations.

```c
bool[] toadd; //initially all false
struct Node {
    int lowner;
    int lnext;
    int[] ticket; //initially all -1
    int data;
    struct Node *next;
}
struct List {
    struct Node *Head;
}
initialize() {
    Head := cons(0, 0, MIN, null);
    Head.next := cons(0, 0, MAX, null);
}
add(e) {
    local p, c, x, pi, co, ci, po, u, r, aux;
    p := Head; toaddcid := true;
    pi := getAndInc(p.lnext); p.ticketpi := cid;
    po := p.lowner;
    while (pi <> po) {
        po := p.lowner;
    }
    c := p.next;
    u := c.data; aux := metric(p);
    while (u < e) {
        ci := getAndInc(c.lnext); c.ticketci := cid;
        co := c.lowner;
        while (ci <> co) {
            co := c.lowner;
        }
        p.lowner++;
        p := c;
        c := p.next;
        u := c.data; aux := metric(p);
    }
    if (u != e) {
        ci := getAndInc(c.lnext); c.ticketci := cid;
        co := c.lowner;
        while (ci <> co) {
            co := c.lowner;
        }
        n := c.next;
        p.next := n;
        dispose(c);
        r := true;
    } else {
        r := false; toaddcid := false;
    }
    return r;
}
rmv(e) {
    local p, c, n, pi, co, ci, po, u, r, aux;
    p := Head;
    <p := Head; toaddcid := true>;
    <pi := getAndInc(p.lnext); p.ticketpi := cid>;
    <po := p.lowner>;
    while (pi <> po) {
        <po := p.lowner>;
    }
    c := p.next;
    <u := c.data; aux := metric(p)>;
    while (u < e) {
        <ci := getAndInc(c.lnext); c.ticketci := cid>;
        <co := c.lowner>;
        while (ci <> co) {
            <co := c.lowner>;
        }
        <p.lowner++>;
        p := c;
        c := p.next;
        <u := c.data; aux := metric(p)>;
    }
    if (u = e) {
        <ci := getAndInc(c.lnext); c.ticketci := cid>;
        <co := c.lowner>;
        while (ci <> co) {
            <co := c.lowner>;
        }
        n := c.next;
        p.next := n;
        dispose(c);
        r := true;
    } else {
        p.lowner++;
        r := false;
    }
    return r;
}
```

Figure 52. Lock-coupling list.
```c
int metric(p)
{
    local n, l, s, o, t;
    n := p.next; l := 0; s := ∅;
    while (n <> null) {
        l++;
        o := n.lowner;
        while (o < n.lnext) {
            t := n.ticket;
            if (toadd_t = true) s := s ∪ {t};
        }
        n := n.next;
    }
    l := l + |s|;
    return l;
}
```

|∅| ≜ 0

|S ∪ {x}| ≜ |S| + 1

len(ε) ≜ 0

len(v::A) ≜ len(A) + 1

toaddThrs(S) ≜ { t | (t ∈ S) ∧ (toadd_t = true)}

Figure 53. Auxiliary code to compute the metric for the list traversal.
\[ A ::= \epsilon \mid v :: A \quad tl ::= \epsilon \mid t :: tl \quad L ::= \epsilon \mid tl + L \quad B \in \text{ThrdD} \rightarrow \text{Bool} \]

\[ I \text{ def } \exists A, L. \text{toadds} * \text{ls}_{L}(\text{Head}, A, \text{null}) * \text{ss}(A) \]

\[ F_l \text{ def } \exists A, L. \text{toadds}(\text{false}) * \text{ls}_{irr_L}(\text{Head}, A, \text{null}) * \text{ss}(A) \]

\[ \text{toadds} \text{ def } \exists B. \text{toadds}(B) \quad \text{toadds}(B) \text{ def } (\oplus_{v} \text{toadd}_{v} = B(t)) \]

\[ \text{toadd}_{l}(b) \text{ def } \exists B. \text{toadd}_{l}(b, B) \quad \text{toadd}_{l}(b, B) \text{ def } (\text{toadds} = b) * (\oplus_{v} \text{toadd}_{v} = B(t')) \]

\[ \text{ls}_{L}(x, A, y) \text{ def } (x = y \wedge A = \epsilon \wedge L = \epsilon \wedge \text{emp}) \]
\[ (x \neq y \wedge \exists z, v, A', \text{tl}, L'). A = v :: A' \wedge L = \text{tl} + L' \wedge \text{N}_{d}(x, v, z) * \text{ls}_{L}((z, A', y)) \]

\[ \text{ls}_{irr_L}(x, A, y) \text{ def } (x = y \wedge A = \epsilon \wedge L = \epsilon \wedge \text{emp}) \]
\[ (x \neq y \wedge \exists z, v, A', \text{tl}, L'). A = v :: A' \wedge L = \text{tl} + L' \wedge \text{N}_{d}(x, v, z) * \text{ls}_{irr_L}((z, A', y)) \]

\[ \text{ls}_{\text{unlocked}}(x, A, y) \text{ def } (x = y \wedge A = \epsilon \wedge \text{emp}) \]
\[ (x \neq y \wedge \exists z, v, A'. A = v :: A' \wedge U(x, v, z) * \text{ls}_{\text{unlocked}}((z, A', y)) \]

\[ \text{N}_{d}(p, v, y) \text{ def } \text{lock}_{d}(p) * (p.\text{data} = v) * (p.\text{next} = y) \]

\[ \text{N}_{d}(p, v, y) \text{ def } \exists n. \text{N}_{d}(p, v, y) \quad \text{L}(p, v, y) \text{ def } \exists n. \text{L}(p, v, y) \]

\[ \text{N}_{d}(p, v, y) \text{ def } \text{lock}_{d}(p) * (p.\text{data} = v) * (p.\text{next} = y) \]

\[ \text{U}(p, v, y) \text{ def } \text{unlockd}(p) * (p.\text{data} = v) * (p.\text{next} = y) \]

\[ \text{lock}_{d}(p) \text{ def } \exists n. \text{lock}_{d}(p) \quad \text{lock}_{d}(p) \text{ def } \text{unlockd}(p) * (tl = \epsilon \wedge \text{unlockd}(p)) \]

\[ \text{lock}_{d}(p) \text{ def } \exists n. \text{lock}_{d}(p) \quad \text{lock}_{d}(p) \text{ def } \text{lock}_{d}(p) \wedge (tl \not\in d) \]

\[ \text{lock}_{d}(p) \text{ def } \exists n. \text{lock}_{d}(p) \quad \text{unlockd}(p) \text{ def } \exists n. \text{unlockd}(p) \]

\[ \text{unlockd}(p) \text{ def } (p.\text{owner} = n) * (p.\text{next} = n) * \text{tickets}(p, 0, n) * \text{tickets}_{\text{new}}(p, n) \]

\[ \text{tickets}(p, n, n) \text{ def } \exists n. \text{tickets}(p, n, n) \]

\[ \text{tickets}(p, n, n) \text{ def } \begin{cases} (n_1 = n_2) \wedge \text{emp} & \text{if } tl = \epsilon \\ (p.\text{ticket}_{n_1} = t) * \text{tickets}(p, n_1 + 1, n_2) & \text{if } tl = t' \end{cases} \]

\[ \text{tickets}_{\text{new}}(p, n) \text{ def } \oplus_{l \geq 2} p.\text{ticket}_{l} = -1 \]

\[ \text{s}(A) \text{ def } \exists A'. (A = \text{MIN} :: A' :: \text{MAX}) \wedge \text{sorted}(A) \]

\[ \text{ss}(A) \text{ def } \exists A'. (A = \text{MIN} :: A' :: \text{MAX}) \wedge \text{sorted}(A) \wedge (B = A') \]

\[ \text{sorted}(A) \text{ def } \begin{cases} \text{true} & \text{if } A = \epsilon \vee A = v :: \epsilon \\ (v_1 < v_2) \wedge \text{sorted}(v_2 :: A') & \text{if } A = v_1 :: v_2 :: A' \end{cases} \]

\[ t \in tl \text{ iff } \exists t', t''. (tl = t' :: t'') \wedge (t = t' \vee t \in tl) \]
\[ R_t \overset{\text{def}}{=} \bigvee_{i \neq t} G'_t \]

\[ G'_t \overset{\text{def}}{=} (\text{ToAdd}_t \lor \text{Add}_t \lor \text{FailedAdd}_t \lor \text{Rmv}_t \lor \text{ReqLock}_t \lor \text{ReqLock}_2 \lor \text{Unlock}_t \lor \text{Id}) \land (I \times I) \]

\[ \text{ToAdd}_t \overset{\text{def}}{=} \exists A, L. [\text{is_irr}_{t,L}(\text{Head}, A, \text{null})] \ast (\text{toadd}_t = \text{false}) \land (\text{toadd}_t = \text{true}) \]

\[ \text{Add}_t \overset{\text{def}}{=} \exists x, y, z, n, v, u, \delta, \delta', n_1, S. (L_{t;:}\cdot,n_1(x, v, y) \ast L_{t;:}\cdot(y, u, z) \ast (S = S) \land (u < w < u)) \]

\[ \text{FailedAdd}_t \overset{\text{def}}{=} (\text{toadd}_t = \text{true}) \land (\text{toadd}_t = \text{false}) \]

\[ \text{Rmv}_t \overset{\text{def}}{=} \exists x, y, z, n, v, u, \delta, \delta', n_1. (L_{t;:}\cdot,n_1(x, v, y) \ast L_{t;:}\cdot(y, u, z) \ast (S = S) \land (u < w < u)) \]

\[ \text{ReqLock}_t \overset{\text{def}}{=} \exists x, y, A, L, \delta, \delta', n_1. (N_{\text{irr}_{t,L},n_1}(\text{Head}, A, x) \lor L_{t;:}\cdot(y, u, z) \lor (u < \text{MAX}) \land (S = S)) \]

\[ \text{Unlock}_t \overset{\text{def}}{=} \exists x, y, A, L, \delta, \delta', n_1. (N_{\text{irr}_{t,L},n_1}(\text{Head}, A, x) \lor L_{t;:}\cdot(y, u, z) \lor (u < \text{MAX}) \land (S = S)) \]

\[ \mathcal{D}_t \overset{\text{def}}{=} \forall x, n_1. dp_r(x, n_1) \sim dq_r(x, n_1) \]

\[ dp_r(x, n_1) \overset{\text{def}}{=} \exists y, z, A, \delta, \delta', L. \text{toadds} \ast N_{\text{irr}_{t,L},n_1}(\text{Head}, A, x) \ast L_{t;:}\cdot(x, v, y) \ast (u < \text{MAX}) \land (S = S) \land (A = A) \]

\[ dq_r(x, n_1) \overset{\text{def}}{=} \exists y, z, A, \delta, \delta', L. \text{toadds} \ast N_{\text{irr}_{t,L},n_1}(\text{Head}, A, x) \ast L_{t;:}\cdot(x, v, y) \ast (u < \text{MAX}) \land (S = S) \land (A = A) \]

\textbf{Figure 55.} Rely, guarantee and definite actions of the lock-coupling list.
(toadd, L) < (toadd', L') iff one of the following holds:

1. \( \text{idset}(L) \subseteq \text{idset}(L') \), or
2. \( \exists S. (\text{idset}(L) = \text{idset}(L') = S) \land (\text{toaddThrds}(S, \text{toadd}) \subseteq \text{toaddThrds}(S, \text{toadd}')) \), or
3. \( \exists S. (\text{idset}(L) = \text{idset}(L') = S) \land (\forall t \in S, (\text{toadd}(t) = \text{toadd}'(t))) \land (\sum_{t \in S} \text{backpos}(t, L) < \sum_{t \in S} \text{backpos}(t, L'))

(\text{toadd}, L) \leq (\text{toadd}', L') iff

\( \exists S. (\text{idset}(L) = \text{idset}(L') = S) \land (\forall t \in S, (\text{toadd}(t) = \text{toadd}'(t))) \land (\sum_{t \in S} \text{backpos}(t, L) = \sum_{t \in S} \text{backpos}(t, L'))

\( \text{idset}(\varepsilon) \overset{\text{def}}{=} \emptyset \quad \text{idset}(\varepsilon + L) \overset{\text{def}}{=} \text{list2set}(\varepsilon) \cup \text{idset}(L) \)

\( \text{list2set}(\varepsilon) \overset{\text{def}}{=} \emptyset \quad \text{list2set}(\varepsilon :: t) \overset{\text{def}}{=} \{ t \} \cup \text{list2set}(t) \)

\( \text{toaddThrds}(S, \text{toadd}) \overset{\text{def}}{=} \{ t \mid (t \in S) \land (\text{toadd}(t) = \text{true}) \}

\( \text{backpos}(t, \varepsilon) \overset{\text{def}}{=} 0 \quad \text{backpos}(t, \varepsilon + L) \overset{\text{def}}{=} \left\{ \begin{array}{ll}
\text{len}(L) & \text{if } t \in L \\
\text{backpos}(t, L) & \text{if } t \not\in L
\end{array} \right.

\( \text{len}(\varepsilon) \overset{\text{def}}{=} 0 \quad \text{len}(\varepsilon + L) \overset{\text{def}}{=} 1 + \text{len}(L) \)

\[
f_1(\emptyset) = (\text{toadd}, L) \quad \text{iff} \quad \emptyset \models J_1(\text{toadd}, L)
\]

\[
J_1(\text{toadd}, L) \overset{\text{def}}{=} \exists z, A, t, t', L'. (L = t + L') \land \text{toadd}(\text{toadd}) \land L_{\text{tciad}, t'}(\text{Head}, \text{MIN}, t) \land L_{z, A, t} \land L_{A, t, t', L} \land L_{t + L}(t)
\]

\( J_1 \overset{\text{def}}{=} \exists B, L_0. J_1(B, L_0) \)

\( Q_1 \overset{\text{def}}{=} \text{Enabled}(D_{\text{ciad}}) \)

\[
f_2(\emptyset) = (\text{toadd}, L) \quad \text{iff} \quad \emptyset \models J_2(\text{toadd}, L)
\]

\[
J_2(\text{toadd}, L) \overset{\text{def}}{=} \exists p, c, z, A_1, v, u, A_2, L_1, t', t'', L_2. (L = t + L_2) \land \text{toadd}(\text{toadd}) \land L_{\text{ciad}, t'}(\text{Head}, A_1, p) \land L_{c, p, v, c} \land L_{z, A, t} \land L_{A, t, t', L} \land L_{t + L}(t)
\]

\( J_2 \overset{\text{def}}{=} \exists B, L_0. J_2(B, L_0) \)

\( Q_2 \overset{\text{def}}{=} \text{Enabled}(D_{\text{ciad}}) \)

\[
f_3(\emptyset) = (\text{toadd}, L) \quad \text{iff} \quad \emptyset \models J_3(\text{toadd}, L)
\]

\[
J_3(\text{toadd}, L) \overset{\text{def}}{=} \exists p, c, z, A_1, v, u, A_2, L_1, t', t'', L_2. (L = t + L_2) \land \text{toadd}(\text{toadd}) \land L_{\text{ciad}, t'}(\text{Head}, A_1, p) \land L_{c, p, v, c} \land (N_{\text{ciad}, A}(c, u, z) \lor L_{A, c, u, z}) \land L_{z, A, t} \land L_{A, t, t', L} \land L_{t + L}(t)
\]

\( J_3 \overset{\text{def}}{=} \exists B, L_0. J_3(B, L_0) \)

\( Q_3 \overset{\text{def}}{=} \text{Enabled}(D_{\text{ciad}}) \)

\( G_3 \overset{\text{def}}{=} (\text{ReqLock2}_{\text{ciad}} \lor \text{Unlock2}_{\text{ciad}} \lor \text{Id}) \land (I \equiv I) \)

\( \text{Unlock2}_{\text{ciad}} \overset{\text{def}}{=} \exists x, y, z, A, v, u, L, t, t', n_{1, 1}, [L_{x, v, y}] \land (L_{x, v, y} \land (x, y) \land (u < \text{MAX})) \)

We can prove:

\( J_1 \Rightarrow (\text{Req}, I): D_{\text{ciad}} \overset{f_1}{\rightarrow} Q_1 \)

\( J_2 \Rightarrow (\text{Req}, I): D_{\text{ciad}} \overset{f_2}{\rightarrow} Q_2 \)

\( J_3 \Rightarrow (\text{Req}, G_3): D_{\text{ciad}} \overset{f_3}{\rightarrow} Q_3 \)

---

**Figure 56.** The well-founded order and the metrics for the loops.
\[
\text{tl}_{d_1,t,d_2,n_1,n}(p,v) \overset{\text{def}}{=} \text{locked}_{d_1,t,d_2,n_1,n}(p) \ast (p.data = v) \ast (p.next = y)
\]

\[
\text{locked}_{d_1,t,d_2,n_1,n}(p) \overset{\text{def}}{=} \exists \pi_1, \pi, \pi_2 \overset{\text{def}}{=}
\]

\[
\begin{align*}
& \exists n_1, t \overset{\text{def}}{=} t \neq t_2 \land (t \neq t_2) \wedge ((p.lowner = n_1) \ast (p.lnext = n_2) \wedge (n_1 \leq n_2)) \\
& \ast \text{tickets}(p, 0, n_1) \ast \text{tickets}_{\pi_1}(p, n_1, n) \ast \text{tickets}_{\pi_2}(p, n_1, n) \ast \text{tickets}_{\text{new}}(p, n_1, n)
\end{align*}
\]

\[
P_1 \overset{\text{def}}{=} \exists z, A, d, d', L, n_1, n. \text{toadd}_{d,d',\pi_1}(\text{Head}, \text{MIN}, z) \ast \text{is irr}_{d,d',l}(z, A, \text{null}) \ast \text{ss}(\text{MIN} \cdot A) \\
\wedge (\text{MIN} < e < \text{MAX}) \wedge (p = \text{Head}) \wedge (po \leq n_1) \wedge (pi = n)
\]

\[
P_1' \overset{\text{def}}{=} \exists z. P_1^z(z)
\]

\[
P_1'' \overset{\text{def}}{=} \exists z, A, d, d', L, n_1, n. \text{toadd}_{d,d',\pi_1}(\text{Head}, \text{MIN}, z) \ast \text{is unlocked}(z, A, \text{null}) \ast \text{ss}(\text{MIN} \cdot A) \\
\wedge (\text{MIN} < e < \text{MAX}) \wedge (p = \text{Head}) \wedge (po \leq n_1) \wedge (pi = n_1)
\]

\[
p_1' \overset{\text{def}}{=} P_1 \wedge \Diamond (pi = po) \lor P_1'' \wedge \Diamond (pi = (po + 1)) \wedge (pi \neq po) \land \text{arem}(\text{ADD}) \land (e = E)
\]

\[
\begin{align*}
\text{add}(e) \{ \\
& \text{local } p, c, x, \pi, \text{po}, \text{ci}, \text{co}, u, r, \text{aux}; \\
& \{ p (\text{MIN} < e < \text{MAX}) \land \text{arem}(\text{ADD}) \land (e = E) \}
\end{align*}
\]

\[
\begin{align*}
& 1 \text{ local } p, c, x, \pi, \text{po}, \text{ci}, \text{co}, u, r, \text{aux}; \\
& \{ p (\text{MIN} < e < \text{MAX}) \land \text{arem}(\text{ADD}) \land (e = E) \}
\end{align*}
\]

\[
\begin{align*}
& 2 <p := \text{Head}; \text{toadd}_{d,d'} := \text{true}; \\
& \{ p (\text{MIN} < e < \text{MAX}) \land \text{arem}(\text{ADD}) \land (e = E) \}
\end{align*}
\]

\[
\begin{align*}
& 3 <\pi := \text{getIdInc}(p.lnext); \text{p.ticket}_{\pi_1} := \text{cid}; \\
& \{ p (\text{MIN} < e < \text{MAX}) \land \text{arem}(\text{ADD}) \land (e = E) \}
\end{align*}
\]

\[
\begin{align*}
& 4 <p := p.lowner; \\
& \{ p (\text{MIN} < e < \text{MAX}) \land \text{arem}(\text{ADD}) \land (e = E) \}
\end{align*}
\]

\[
\begin{align*}
& 5 \text{ while } (pi < po) \{ \\
& \{ p \}
\end{align*}
\]

\[
\begin{align*}
& 6 <p := p.lowner; \\
& \{ p (\text{MIN} < e < \text{MAX}) \land \text{arem}(\text{ADD}) \land (e = E) \}
\end{align*}
\]

\[
\begin{align*}
& 7 \} \\
& \{ p' \land \text{arem}(\text{ADD}) \land (e = E) \}
\end{align*}
\]

\[
\begin{align*}
& 8 <c := p.next; \\
& \{ p' \land \text{arem}(\text{ADD}) \land (e = E) \}
\end{align*}
\]

\[
\begin{align*}
& 9 \ldots
\end{align*}
\]

**Figure 57.** Proof outline for add (1).
\[
\begin{align*}
\text{\underline{Proof outline for add (2):}}
\end{align*}
\]
\[ P_3 \triangleq \]
\[ \exists z, A_1, v, A_2, L_1, h, h', L_2, n_1, n. \text{toaddc}(\text{true}) \times \text{lsirr}_{cid,l_1}(\text{Head}, A_1, p) \times L_{cid:df}(p, v, c) \]
\[ \times \text{tl}_{cid,n_1}(c, u, z) \times \text{lsirr}_{cid,l_2}(z, A_2, \text{null}) \times \text{ss}(A_1 \vdash v::u::A_2) \land (u < e < \text{MAX}) \]
\[ \land (c_1 = n_1) \land (1 + \text{len}(A_2) + |\text{toaddThds}(\text{tidset}(h' + L_2)|) \leq \text{aux}) \]

\[ P_3' \triangleq \]
\[ \exists z, A_1, v, A_2, L_1, h, L_2. \text{toaddc}(\text{true}) \times \text{lsirr}_{cid,l_1}(\text{Head}, A_1, p) \times L_{cid:df}(p, v, c) \]
\[ \times \text{lsirr}_{cid,n_1}(c, u, z) \times \text{lsirr}_{cid,l_2}(z, A_2, \text{null}) \times \text{ss}(A_1 \vdash v::u::A_2) \land (u < e < \text{MAX}) \]
\[ \land (c_1 = n_1) \land (1 + \text{len}(A_2) + |\text{toaddThds}(\text{tidset}(L_2))|) \leq \text{aux}) \]

\[ P_3'' \triangleq \]
\[ \exists z, A_1, v, A_2, L_1, L_2, n, n_1. \text{toaddc}(\text{true}) \times \text{lsirr}_{cid,l_1}(\text{Head}, A_1, p) \times L_{cid:df}(p, v, c) \]
\[ \times \text{lsirr}_{cid,n_1}(c, u, z) \times \text{lsirr}_{cid,l_2}(z, A_2, \text{null}) \times \text{ss}(A_1 \vdash v::u::A_2) \land (u < e < \text{MAX}) \]
\[ \land (c_1 = n_1) \land (c_1 = n_1) \land (1 + \text{len}(A_2) \leq \text{aux}) \]

\[ \{ p_2 \} \]
\[ \{ P_2 \land \text{arem}(\text{ADD}) \land (e = E) \land \text{O}(c_1 - co) \} \]
\[ \text{while} (c_1 < co) \{ \]
\[ \{ (p_3 \land \text{O}(c_1 - c_0) \lor P'' \land \text{O}(c_1 - (c_1 + 1))) \land (c_1 \neq c_0) \land \text{arem}(\text{ADD}) \land (e = E) \} \]
\[ \text{<c0} := \text{c0} \text{.lwren}; \]
\[ \{ P_2 \land \text{arem}(\text{ADD}) \land (e = E) \land \text{O}(c_1 - co) \} \]
\[ \{ P'' \land \text{arem}(\text{ADD}) \land (e = E) \} \]
\[ \{ p_3 \} \]

\[ \{ P_3 \land (e \leq u) \land \text{arem}(\text{ADD}) \land (e = E) \} \]
\[ \exists z, A_1, v, A_2, L_1, h, L_2. \text{toaddc}(\text{true}) \times \text{lsirr}_{cid,l_1}(\text{Head}, A_1, p) \times L_{cid:df}(p, v, c) \]
\[ \times \text{lsirr}_{cid,n_1}(c, u, z) \times \text{lsirr}_{cid,l_2}(z, A_2, \text{null}) \times \text{ss}(A_1 \vdash v::u::A_2) \land (v < u \leq u) \land (e < \text{MAX}) \land \text{arem}(\text{ADD}) \land (e = E) \}
\]

\[ \text{if} (u = e) \{ \]
\[ \exists z, A_1, v, A_2, L_1, h, L_2. \text{toaddc}(\text{true}) \times \text{lsirr}_{cid,l_1}(\text{Head}, A_1, p) \times L_{cid:df}(p, v, c) \]
\[ \times \text{lsirr}_{cid,n_1}(c, u, z) \times \text{lsirr}_{cid,l_2}(z, A_2, \text{null}) \times \text{ss}(A_1 \vdash v::u::A_2) \land (v < e < u) \land \text{arem}(\text{ADD}) \land (e = E) \}
\]

\[ x := \text{cons}(0, v, e, c); \]
\[ \exists z, A_1, v, A_2, L_1, h, L_2. \text{toaddc}(\text{true}) \times \text{lsirr}_{cid,l_1}(\text{Head}, A_1, p) \times L_{cid:df}(p, v, c) \land U(x, e, c) \]
\[ \times \text{lsirr}_{cid,n_1}(c, u, z) \times \text{lsirr}_{cid,l_2}(z, A_2, \text{null}) \times \text{ss}(A_1 \vdash v::u::A_2) \land (v < e < u) \land \text{arem}(\text{ADD}) \land (e = E) \}
\]

\[ \text{<p. next} := x; \text{toaddc} := \text{false}; \]
\[ \exists z, A_1, v, A_2, L_1, h, L_2. \text{toaddc}(\text{false}) \times \text{lsirr}_{cid,l_1}(\text{Head}, A_1, p) \times L_{cid:df}(p, v, x) \]
\[ \times \text{lsirr}_{cid,n_1}(x, e, c) \times \text{lsirr}_{cid,l_2}(x, A_2, \text{null}) \times \text{ss}(A_1 \vdash v::e::u::A_2) \land \text{arem}(\text{skip}) \land (R = \text{true}) \]
\[ r := \text{true}; \]
\[ \text{<x := false; \text{toaddc} := \text{false};} \]
\[ \text{if} \]
\[ \exists z, A_1, v, A_2, L_1, h, L_2. \text{toaddc}(\text{false}) \times \text{lsirr}_{cid,l_1}(\text{Head}, A_1, p) \times L_{cid:df}(p, v, z) \]
\[ \times \text{lsirr}_{cid,l_2}(z, A_2, \text{null}) \times \text{ss}(A_1 \vdash v::u::A_2) \land \text{arem}(\text{skip}) \land (r = \text{R}) \]
\[ \text{<p. loowner++;} \]
\[ \exists A, L. \text{toaddc}(\text{false}) \times \text{lsirr}_{cid,l}(\text{Head}, A, \text{null}) \times \text{ss}(A) \land \text{arem}(\text{skip}) \land (r = \text{R}) \}
\[ \{ P \land \text{arem}(\text{skip}) \land (r = \text{R}) \} \]
\[ \text{return} \ r; \]

---

**Figure 59.** Proof outline for add (2) – the loop at lines 13–15.

**Figure 60.** Proof outline for add (3).
\[
\text{rmv}(e) \{ \\
1 \text{ local p, c, n, pi, po, ci, co, u, r, aux; } \\
\{ P \land (\text{MIN} < e < \text{MAX}) \land \text{arem(RMV)} \land (e = E) \} \\
2 \text{ p := Head; } \\
\ldots \\
9 \text{ u := c.data; aux := metric(p);} \\
10 \text{ while (u < e) } \{ \\
11 \text{ if (u = e) } \{ \\
12 \text{ rmv(e) } \\
13 \text{ ci := getAndInc(c.lnext); c.ticket} \\
14 \text{ co := c.lowner;} \\
15 \text{ r := true; } \\
16 \text{ r := false; } \\
17 \text{ } \}
18 \text{ u := c.data; aux := metric(p);} \\
19 \text{ while (u < e) } \{ \\
20 \text{ if (u = e) } \{ \\
21 \text{ rmv(e) } \\
22 \text{ ci := getAndInc(c.lnext); c.ticket} \\
23 \text{ co := c.lowner;} \\
24 \text{ r := true; } \\
25 \text{ r := false; } \\
26 \} \\
27 \text{ } \}
28 \text{ p.next := n; } \\
29 \text{ } \}
30 \text{ dispose(c);} \\
31 \text{ r := true; } \\
32 \text{ else } \{ \\
33 \text{ } \}
34 \text{ return r; } \\
\}
\]

Figure 61. Proof outline for \text{rmv}. 

94
\( P_3 \equiv \exists z, A_1, v, A_2, L_1, ll', L_2, n_1, n. \ \text{toadd}_{cid}(false) * \text{ls}_{irr}_{cid}_{L_1}(\text{Head}, A_1, p) * L_{cid}_{\text{id}}(p, v, c) * L_{L_1, \text{id}}(z, e, z) * \text{ls}_{irr}_{cid}_{L_2}(z, A_2, null) * \text{ss}(A_1 : v : e : A_2) * (e < \text{MAX}) * (co \leq n_1) * (ci = n) \)

\( P_4 \equiv \exists z, A_1, v, A_2, L_1, ll, L_2, n_1, n. \ \text{toadd}_{cid}(false) * \text{ls}_{irr}_{cid}_{L_1}(\text{Head}, A_1, p) * L_{cid}_{\text{id}}(p, v, c) * L_{L_1, \text{id}}(z, e, z) * \text{ls}_{irr}_{cid}_{L_2}(z, A_2, null) * \text{ss}(A_1 : v : e : A_2) * (e < \text{MAX}) \)

\( P_4' \equiv \exists z, A_1, v, A_2, L_1, ll, n_1. \ \text{toadd}_{cid}(false) * \text{ls}_{irr}_{cid}_{L_1}(\text{Head}, A_1, p) * L_{cid}_{\text{id}}(p, v, c) * L_{L_1, \text{id}}(z, A_2, null) * \text{ss}(A_1 : v : e : A_2) * (e < \text{MAX}) * (co \leq n_1) * (ci = n_1) \)

\((P_4 \land \text{arem}(\text{RMV}) \land (e = E) \land (ci - co) \land (ci \neq co) \land Q_2 \land \text{true}) \implies (P_4' \land (ci \neq co) \land \text{arem}(\text{RMV}) \land (e = E) \land (ci - (co + 1))) \land (\langle 1 \rangle \land \text{emp}) \)

Also remember \( J_2 \Rightarrow (R_{cid}, \{I\} : D_{cid} \xrightarrow{I} Q_2) \)

\[
\{ P_4 \land \text{arem}(\text{RMV}) \land (e = E) \} \\
\{ P_4 \land \text{arem}(\text{RMV}) \land (e = E) \land (ci - co) \}
\]

24 while (ci <> co) {
    \[
    \{(P_4 \land (ci - co) \lor P_4' \land (ci - (co + 1))) \land (ci \neq co) \land \text{arem}(\text{RMV}) \land (e = E) \}
    \]
25     <co := c.lowner>; 
    \[
    \{ P_4 \land \text{arem}(\text{RMV}) \land (e = E) \land (ci - co) \}
    \]
26 } 
\[
\{ P_4' \land \text{arem}(\text{RMV}) \land (e = E) \}
\]

\textbf{Figure 62.} Proof outline for rnv – the loop at lines 24–26.
D. Deadlock-free examples

In this section, we apply LiLi to verify objects implemented with test-and-set (TAS) locks.

D.1 Counter with TAS lock

The implementation ofInc is given in Fig. 1(b). Below we prove that it is linearizable with respect to the atomic operation INC and is also starvation-free.

At the left of Fig. 63, we give the specification of the code; and at the right of the figure we show the proof outline. The proofs are similar to the proof of the TAS lock in Sec. 4.3.2, though now we also take into account the resource of the counters x and X being protected.

\[
\begin{align*}
\text{I} & \equiv \text{unlocked} \ast \text{resource} \lor \text{locked} \\
\text{resource} & \equiv (x = X) \\
\text{unlocked} & \equiv (L = 0) \\
\text{locked} & \equiv \exists t. \text{locked}(t) \\
\text{locked}(t) & \equiv (L = t) \land (t \in \text{TIDS}) \\
\text{envLocked}(t) & \equiv \exists t'. \text{locked}(t') \land (t' \neq t) \\
\text{P}_{cid} & \equiv \text{unlocked} \ast \text{resource} \lor \text{envLocked}(cid) \\
\text{G}_{cid} & \equiv (\text{Lock}_{cid} \lor \text{Unlock}_{cid} \lor \text{Id}) \ast \text{Id} \land (I \equiv I) \\
\text{Lock}_{cid} & \equiv (\text{unlocked} \ast \text{resource}) \land (\text{locked}(cid) \lor \text{P}_{cid}) \\
\text{Unlock}_{cid} & \equiv \text{locked}(cid) \land \text{unlocked}(cid) \\
\text{D}_{cid} & \equiv \text{locked}(cid) \land (\text{unlocked} \ast \text{resource}) \\
\end{align*}
\]

\[
\begin{align*}
p & \equiv (\text{locked}(cid) \ast \text{resource} \land b) \lor (P \land \neg b) \land \Diamond (1) \land \Box (1)) \\
p' & \equiv (\text{unlocked} \ast \text{resource} \land \Box (1) \land \Box (0)) \lor (\text{envLocked}(cid) \land \Diamond (1) \land \Box (1)) \\
Q & \equiv \text{locked}(cid) \lor \text{unlocked} \ast \text{resource} \\
J & \equiv \text{locked}(cid) \lor P \\
\end{align*}
\]

\[
\begin{align*}
inc(): \\
1 & \text{local } b, r; \{ P \land \Diamond (1) \land \text{arem(INC)} \} \\
2 & b := \text{false}; \{ p \land \text{arem(INC)} \} \\
3 & \text{while }(!b) \{ \{ p' \land \text{arem(INC)} \} \\
4 & b := \text{cas}(\text{L}, 0, cid); \{ p \land \text{arem(INC)} \} \\
5 & \} \{ \text{locked}(cid) \ast \text{resource} \land \text{arem(INC)} \} \\
6 & r := x; x := r + 1; \{ \text{locked}(cid) \ast (x = X + 1) \land \text{arem(INC)} \} \\
7 & L := 0; \{ P \land \text{arem(skip)} \} \\
\end{align*}
\]

Figure 63. Proof outline of the counter with TAS lock.
D.2 Two-lock queue with TAS lock

Below we prove that the two-lock queue implemented with TAS locks is deadlock-free. The implementation is shown in Fig. 64.

The proofs are similar to the proofs in Sec. 1C.5 where the locks are implemented using ticket locks. Fig. 65 defines the precise invariant, the precondition, the rely and guarantee conditions and the definite action. Fig. 66 and Fig. 67 show the proof outlines. Note that the proofs for the loops of acquiring the locks follow the proofs for the counter with TAS locks (see Sec. 1D.1).

```
enq(v) {
    local x, b;
    x := cons(v, null);
    b := false;
    while (!b) {
        b := cas(&Tlock, 0, cid);
    }
    Tail.next := x;
    Tail := x;
    Tlock := 0;
}
```

```
deq() { 
    local h, s, v, b;
    b := false;
    while (!b) {
        b := cas(&Hlock, 0, cid);
    }
    h := Head;
    s := h.next;
    if (s = null) {
        Hlock := 0;
        v := EMPTY;
    } else {
        v := s.data;
        Head := s;
        Hlock := 0;
        dispose(h);
    }
    return v;
}
```

Figure 64. Two-lock queue implementation with TAS lock.
\[ I \equiv \exists h, z, s. (\text{Head} = h) \land (\text{Tail} = z) \land \text{queue}(s, h, z) \land \text{lock}(h) \land \text{lock}(s) \]

\[ \text{queue}(s, h, z) \equiv \exists v, A. (Q = A) \land (\text{unlag}(h, z, v) \lor \text{lag}(h, z, v) \land (s \neq 0)) \lor (\text{cross}(h, v) \land (s \neq 0)) \]

\[ \text{unlag}(h, z, A) \equiv \exists v, A'. (A = A' \lor \text{ls}(h, A', z) \land N(z, v, \text{null})) \]

\[ \text{lag}(h, z, A \lor \text{lag}(h, z, v) \land \text{ls}(h, A', z) \land N(z, v, \text{null})) \]

\[ \text{cross}(h, A) \equiv \exists v. (A = v) \land N(h, v, \text{null}) \]

\[ \text{ls}(x, A, y) \equiv (x = y \land A = e) \lor (x \neq y \land \exists z, v, A'. A = v \land N(x, v, z) \land \text{ls}(z, A', y)) \]

\[ N(p, v, y) \equiv (p.\text{data} = v) \lor (p.\text{next} = y) \]

\[ \text{lock}(l) \equiv \exists s. \text{lock}_s(l) \]

\[ \text{lock}_s(l) \equiv \begin{cases} (l = s) & \text{lock}_s(l) \equiv \exists s. \text{lock}_s(l) \land \text{lock}_s(l) \equiv \text{lock}_s(l) \land (s \neq t) \end{cases} \]

\[ \text{unlocked}(l) \equiv \text{lock}_0(l) \land \text{locked}(l) \equiv \exists t. \text{locked}(l) \land \text{locked}(l) \equiv \text{lock}(l) \land (t \in \text{TIDS}) \]

\[ P_t \equiv \exists h, z, s. (\text{Head} = h) \land (\text{Tail} = z) \land \text{queue}(s, h, z) \land \text{lock}(h) \land \text{lock}_t(s) \]

\[ R_t \equiv \exists h, z, s. (\text{Head} = h) \land (\text{Tail} = z) \land \text{queue}(s, h, z) \land \text{lock}(h) \land \text{lock}_t(s) \]

\[ G_t \equiv (\text{Enq}_t \lor \text{Swing}_t \lor \text{Deq}_t \lor \text{LockH}_t \lor \text{UnlockH}_t \lor \text{LockT}_t \lor \text{UnlockT}_t \lor \text{Id}) \land (I \land I) \]

\[ \text{Enq}_t \equiv \begin{cases} \forall x, y, A, v. [\text{locked}(l) \land (\text{Tail} = x)] \land (N(x, v, y, v) \land (Q = A)) \land (N2(x, v, y, v, \text{null}) \land (Q = A)) \end{cases} \]

\[ \text{Swing}_t \equiv \begin{cases} \forall x, v. [\text{locked}(l) \land N(x, v, \text{null})] \land (\text{Tail} = x) \end{cases} \]

\[ \text{Deq}_t \equiv \begin{cases} \exists x, y, z, v, w, \text{A}. [\text{locked}(l) \land (\text{Tail} = x)] \land (\text{Head} = x) \land N2(x, v, y, v, z) \land (Q = A) \end{cases} \]

\[ \text{LockH}_t \equiv \begin{cases} \text{lock}(h) \land (\text{Head} = h) \land \text{locked}(l) \land (\text{Locked} = l) \end{cases} \]

\[ \text{UnlockH}_t \equiv \begin{cases} \text{lock}(h) \land (\text{Head} = h) \land \text{unlocked}(l) \land (\text{Locked} = l) \end{cases} \]

\[ \text{LockT}_t \equiv \begin{cases} \text{lock}(h) \land (\text{Head} = h) \land \text{locked}(l) \land (\text{Locked} = l) \end{cases} \]

\[ \text{UnlockT}_t \equiv \begin{cases} \text{lock}(h) \land (\text{Head} = h) \land \text{unlocked}(l) \land (\text{Locked} = l) \end{cases} \]

\[ D_t \equiv (dpH_t \sim dqH_t) \land (dpT_t \sim dqT_t) \]

\[ dpH_t \equiv \text{locked}(h) \land \text{true} \land I \quad dqH_t \equiv \text{unlocked}(h) \land \text{true} \land I \]

\[ dpT_t \equiv \text{locked}(l) \land \text{true} \land I \quad dqT_t \equiv \text{unlocked}(l) \land \text{true} \land I \]

**Figure 65.** Invariant, precondition, rely/guarantee and definite actions of the two-lock queue with TAS locks.
enq(v) {
  1 local x, b;
  2 P \lor \text{arem(ENQ)} \land (v = \text{null}) \land (1)
  3 x := \text{cons}(v, \text{null});
  4 while (!b) {
    5 b := \text{cas}(&Hlock, 0, cid);
    6 }
  7 \text{Tail.next} := x;
  8 \text{Tail} := x;
  9 \text{Tlock} := 0;
  10 \{ P \land \text{arem(skip)} \}
}

\text{Figure 66. Proof outline for enq.}

\begin{align*}
P_e^{\text{loc}} & \overset{\text{def}}{=} \\
& \exists h, z, v_d. \text{(Head} = h) \land \text{(Tail} = h) \land (Q = A) \land \text{unlag}(h, z, v_d \cdot A) \land \text{lock.irr}(Hlock) \land \text{locked}(Tlock) \\
P_e^{(\text{loc})} & \overset{\text{def}}{=} \\
& \exists h, z. \text{(Head} = h) \land \text{(Tail} = h) \land (Q = A) \land \text{unlag}(h, z, A) \land \text{lag}(x, z, A) \land (s \neq 0) \land \text{locked}(Hlock) \land \text{locked}(Tlock) \\
P_e^{(\text{loc})} & \overset{\text{def}}{=} \\
& \exists h, z. \text{(Head} = h) \land \text{(Tail} = h) \land (Q = \text{null}) \land \text{N}(h, z, v_d) \land \text{locked}(Hlock) \land \text{locked}(Tlock) \\
P_e^{(\text{loc})} & \overset{\text{def}}{=} \\
& \exists h, z, v, A. \text{(Head} = h) \land \text{(Tail} = h) \land (Q = A) \land \text{N}(h, z, v) \land \text{locked}(Hlock) \land \text{locked}(Tlock) \\
\end{align*}

\text{int deq}() {
  1 local h, s, v, b;
  2 b := \text{false};
  3 while (!b) {
    4 b := \text{cas}(&Hlock, 0, cid);
    5 }
  6 h := \text{Head};
  7 \{ P_e^{(\text{loc})} \land \text{arem(DEQ)} \}
  8 a := h.next;
  9 \{ ((s = \text{null}) \land P_e^{(\text{loc})} \land \text{arem(skip)} \land (V = \text{null})) \lor (P_e^{(\text{loc})} \land \text{arem(DEQ)}) \}
  10 if (s = \text{null}) {
    11 \{ P_e^{(\text{loc})} \land \text{arem(skip)} \land (V = \text{null}) \}
    12 Hlock := 0;
    13 \{ P \land \text{arem(skip)} \land (V = \text{null}) \}
  14 v := \text{null};
  15 \{ P \land \text{arem(skip)} \land (V = \text{null}) \}
  16 dispose(h);
  17 return v;
}

\text{Figure 67. Proof outline for deq.}
D.3 Lock-coupling list with TAS lock

Fig. 68 shows the code of the lock-coupling list implemented with TAS locks (where the auxiliary code for computing the metric of the list traversal is shown in Fig. 69). We prove its deadlock-freedom.

Fig. 70 defines the precise invariant and the precondition. Fig. 71 defines the rely/guarantee conditions and the definite actions. These definitions are similar to the ones for verifying the lock-coupling list with ticket locks (Appendix C.6). Other figures in this section give the proof outlines.

In the proofs, the only delaying action of a thread \( t \) is that \( t \) acquires the lock of the head node in the list. Thus each method is given one \( \mathcal{E} \)-token only. After acquiring the lock of the head node, the threads doing the operations stand in line. As a result, the list implementation will become starvation-free so long as the lock of the head node is implemented using a ticket lock or a queue lock (even if the locks of other nodes are implemented using TAS locks).

```c
bool[] toadd; //initially all false

struct Node {
    int lock;
    int data;
    struct Node *next;
}

struct List {
    struct Node *Head;
}

initialize(){
    Head := cons(0, MIN, null);
    Head.next := cons(0, MAX, null);
}

add(e) {
    1 local p, c, x, b, u, r, aux;
    2 p := Head; toadd := true;
    3 b := false;
    4 while (!b) {
        5 b := cas(p.lock, 0, cid);
    }
    7 c := p.next;
    8 u := c.data; aux := metricTAS(p);  
    9 while (u < e) {
        10 b := false;
        11 while (!b) {
            12 b := cas(c.lock, 0, cid);
        }
        14 p.lock := 0;
        15 p := c;
        16 c := p.next;
        17 u := c.data; aux := metricTAS(p);  
    }  
    19 if (u != e) {
        20 x := cons(0, e, c);
        21 <p.next := x; toadd := false;>
        22 r := true;
    } else {
        24 <x := false; toadd := false;>
        25 }  
    26 p.lock := 0;
    27 return r;
}

rmv(e) {
    1 local p, c, n, b, u, r, aux;
    2 p := Head;
    3 b := false;
    4 while (!b) {
        5 b := cas(p.lock, 0, cid);
    }
    7 c := p.next;
    8 u := c.data; aux := metricTAS(p);  
    9 while (u < e) {
        10 b := false;
        11 while (!b) {
            12 b := cas(c.lock, 0, cid);
        }
        14 p.lock := 0;
        15 p := c;
        16 c := p.next;
        17 u := c.data; aux := metricTAS(p);  
    }  
    19 if (u = e) {
        20 b := false;
        21 while (!b) {
            22 b := cas(c.lock, 0, cid);
        }
        24 n := c.next;
        25 p.next := n;
        26 p.lock := 0;
        27 dispose(c);
        28 r := true;
    } else {
        29 return r;
    }
}
```

Figure 68. Lock-coupling list with TAS lock.
Figure 70. Auxiliary code to compute the metric for the list traversal.

Figure 71. Invariant and precondition of the lock-coupling list with TAS lock.
\[
G_{cid} \overset{\text{def}}{=} (ToAdd_{cid} \lor Add_{cid} \lor FailedAdd_{cid} \lor Rmv_{cid} \lor LockH_{cid} \lor Lock2_{cid} \lor Unlock_{cid} \lor \text{Id}) \land (I \land (I \land I))
\]

ToAdd_{i} \overset{\text{def}}{=} \exists A, L, [ls_irr_{L}(\text{Head}, A, null)] \land ((\text{toadd}_i = \text{false}) \land (\text{toadd}_i = \text{true}))

Add_{i} \overset{\text{def}}{=} \exists x, y, z, n, v, w, u, s, S.
\((\text{toadd}_i = \text{true}) \land (S = S) \land (L_i(x, v, y) \land U(y, w, z) \land (\text{toadd}_i = \text{false}) \land (S = S \cup \{u\}) \land (u < w < u)) \land (\text{toadd}_i = \text{true}) \land (\text{toadd}_i = \text{false})})

Locked_{i} \overset{\text{def}}{=} (\text{toadd}_i = \text{true}) \land (\text{toadd}_i = \text{false})

Rmv_{i} \overset{\text{def}}{=} \exists x, y, z, n, v, w, u, s, S. (L_i(x, v, y) \land (S = S \cup \{u\}) \land (u < \text{MAX}) \land (L_i(x, v, z) \land (S = S))

LockH_{i} \overset{\text{def}}{=} \exists x, A, L, (L_i(\text{Head}, MIN, x) \land L_i(\text{Head}, MIN, x) \land [ls_irr_{L}(x, A, null)] \land \text{toadd}_{LS} \land \text{toadd}_{LS})

\text{Lock2}_{i} \overset{\text{def}}{=} \exists x, v, z, A, v, A', L', [ls_irr_{L}(\text{Head}, A, x) \land L_i(x, v, y) \land (U(y, u, z) \land L_i(v, u, z)) \land ([u < \text{MAX}] \land ls_irr_{L}(x, A', null)] \land \text{toadd}_{LS} \land \text{toadd}_{LS})

Unlock_{i} \overset{\text{def}}{=} \exists x, y, A, v, L, [ls_irr_{L}(\text{Head}, A, x)] \land (L_i(x, v, y) \land U(x, v, y))

\mathcal{D}_{cid} \overset{\text{def}}{=} \forall x, d_{pcid}(x) \sim d_{scid}(x)

d_{pc}(x) \overset{\text{def}}{=} \exists y, z, A, v, u, A', \text{toadd} \land ls_irr_{L}(\text{Head}, A, x) \land L_i(x, v, y) \land (U(y, u, z) \land L_i(y, u, z)) \land ls_unlocked(z, A', null) \land ss(A :: v :: u :: A')

d_{sc}(x) \overset{\text{def}}{=} \exists y, z, A, v, u, A', \text{toadd} \land ls_irr_{L}(\text{Head}, A, x) \land U(x, v, y) \land (U(y, u, z) \land L_i(y, u, z)) \land ls_unlocked(z, A', null) \land ss(A :: v :: u :: A')

Figure 71. Rely, guarantee and definite actions of the lock-coupling list with TAS lock.

\((\text{toadd}, L) < (\text{toadd}', L')\) iff one of the following holds:

1. \(\text{tidset}(L) \subset \text{tidset}(L')\), or
2. \(\exists S. (\text{tidset}(L) = \text{tidset}(L') = S) \land (\text{toaddThrs}(S, \text{toadd}) \subset \text{toaddThrs}(S, \text{toadd}'))\), or
3. \(\exists S. (\text{tidset}(L) = \text{tidset}(L') = S) \land (\forall t \in S. (\text{toadd}(t) = \text{toadd}'(t))) \land (\sum_{t \in S} \text{backpos}(t, L) < \sum_{t \in S} \text{backpos}(t, L'))\)

\((\text{toadd}, L) \leq (\text{toadd}', L')\) iff

\(\exists S. (\text{tidset}(L) = \text{tidset}(L') = S) \land (\forall t \in S. (\text{toadd}(t) = \text{toadd}'(t))) \land (\sum_{t \in S} \text{backpos}(t, L) = \sum_{t \in S} \text{backpos}(t, L'))\)

\(\text{tidset}(\epsilon) \overset{\text{def}}{=} \emptyset\)

\(\text{tidset}(t::L) \overset{\text{def}}{=} \{t\} \cup \text{tidset}(L)\)

\(\text{tidset}(0::L) \overset{\text{def}}{=} \text{tidset}(L)\)

\(\text{toaddThrs}(S, \text{toadd}) \overset{\text{def}}{=} \{t \mid (t \in S) \land (\text{toadd}(t) = \text{true})\}\)

\(\text{backpos}(t, \epsilon) \overset{\text{def}}{=} 0\)

\(\text{backpos}(t, s::L) \overset{\text{def}}{=} \begin{cases} \text{len}(L) & \text{if } t = s \\ \text{backpos}(t, L) & \text{if } t \neq s \end{cases}\)

\(\text{len}(\epsilon) \overset{\text{def}}{=} 0\)

\(\text{len}(s::L) \overset{\text{def}}{=} 1 + \text{len}(L)\)

Figure 72. The well-founded order.
$f_1(\emptyset) = (\text{toadd}, L)$ if $\emptyset \models J_1(\text{toadd}, L)$

$J_1(\text{toadd}, L) \overset{\text{def}}{=} \exists z, A, s, L'. (L = s \cdot L')$

$\wedge \text{toadd}(\text{toadd}) * N, \text{(Head, MIN, z)} * \text{is_irr}_{L,t}(z, A, \text{null}) * \text{ss}(\text{MIN} :: A)$

$J_1 \overset{\text{def}}{=} \exists B, L_0. J_1(B, L_0)$

$Q_1 \overset{\text{def}}{=} \exists z, A. \text{toadd} * (\text{L}_{\text{cid}}(\text{Head, MIN}, z) \lor U(\text{Head, MIN}, z)) * \text{is_unlocked}(z, A, \text{null}) * \text{ss}(\text{MIN} :: A)$

$f_2(\emptyset) = (\text{toadd}, L)$ if $\emptyset \models J_2(\text{toadd}, L)$

$J_2(\text{toadd}, L) \overset{\text{def}}{=} \exists p, c, z, A_1, v, u, A_2, L_1, s, L_2. (L = s \cdot L_2)$

$\wedge \text{toadd}(\text{toadd}) * \text{is_irr}_{L,t}(\text{Head, A_1}, p) * \text{L}_{\text{cid}}(p, v, c)$

$\wedge N, (c, u, z) * \text{is_irr}_{L,t}(z, A_2, \text{null}) * \text{ss}(A_1 :: v :: u :: A_2)$

$J_2 \overset{\text{def}}{=} \exists B, L_0. J_2(B, L_0)$

$Q_2 \overset{\text{def}}{=} \exists p, c, z, A_1, v, u, A_2, L_1, \text{toadd} * \text{is_irr}_{L,t}(\text{Head, A_1}, p) * \text{L}_{\text{cid}}(p, v, c)$

$\wedge (U(c, u, z) \lor \text{L}_{\text{cid}}(c, u, z)) * \text{is_unlocked}(z, A_2, \text{null}) * \text{ss}(A_1 :: v :: u :: A_2)$

$G_3 \overset{\text{def}}{=} (\text{Lock}_{2_{\text{cid}}} \lor \text{Unlock}_{2_{\text{cid}}} \lor \text{Id}) \land (I \Rightarrow I)$

We can prove:

$J_1 \Rightarrow (R_{\text{cid}}, I) : D_{\text{cid}} f_1 \rightarrow Q_1$

$J_2 \Rightarrow (R_{\text{cid}}, I) : D_{\text{cid}} f_2 \rightarrow Q_2$

$J_2 \Rightarrow (R_{\text{cid}}, G_3 : D_{\text{cid}} f_2 \rightarrow Q_2$

Figure 73. The metrics.

$p_1 \overset{\text{def}}{=} \exists z, A, L. \text{toadd}_{\text{cid}}(\text{true}) * U(\text{Head, MIN, z}) * \text{is_irr}_{L,t}(z, A, \text{null}) * \text{ss}(\text{MIN} :: A)$ \land (p = \text{Head})

$p_1' \overset{\text{def}}{=} \exists z, A, s, L. \text{toadd}_{\text{cid}}(\text{true}) * \text{is_irr}_{L,t}(\text{Head, MIN, z}) * \text{is_irr}_{L,t}(z, A, \text{null}) * \text{ss}(\text{MIN} :: A)$ \land (p = \text{Head})

$p_1''(z) \overset{\text{def}}{=} \exists z, A, L. \text{toadd}_{\text{cid}}(\text{true}) * \text{L}_{\text{cid}}(\text{Head, MIN, z}) * \text{is_irr}_{L,t}(z, A, \text{null}) * \text{ss}(\text{MIN} :: A)$ \land (p = \text{Head})

add(e) {
  1 local p, c, x, b, u, r, aux;
  2 \{ P :: (1) \wedge \text{arem}(ADD) \wedge (\text{MIN} < e < \text{MAX}) \wedge (e = E) \}
  3 <p := \text{Head; toadd}_{\text{cid}} := \text{true}>;
  4 \{ (p_1 \lor p_1') \wedge \text{arem}(ADD) \wedge (\text{MIN} < e < \text{MAX}) \wedge (e = E) \}
  5 \text{b} := \text{false};
  6 \{ (p_1'' \wedge b) \lor (p_1 \lor p_1') \wedge b \wedge (\text{null} :: (1) \wedge \text{are}(0)) \wedge \text{arem}(ADD) \wedge (\text{MIN} < e < \text{MAX}) \wedge (e = E) \}
  7 \text{while} \{ !b \{
  8 \{ (p_1'' \wedge \text{null} :: (1) \wedge \text{null} :: (0)) \lor (p_1 \lor p_1') \wedge \text{null} :: (1) \wedge \text{null} :: (0)) \wedge \text{arem}(ADD) \wedge (\text{MIN} < e < \text{MAX}) \wedge (e = E) \}
  9 \text{c} := \text{p.next};
  10 \{ p_1''(c) \wedge \text{arem}(ADD) \wedge (\text{MIN} < e < \text{MAX}) \wedge (e = E) \}
  11 \}
  12 \{ p_1'' \wedge \text{arem}(ADD) \wedge (\text{MIN} < e < \text{MAX}) \wedge (e = E) \}
  13 \text{c} := \text{p.next};
  14 \{ p_1''(c) \wedge \text{arem}(ADD) \wedge (\text{MIN} < e < \text{MAX}) \wedge (e = E) \}
  15 \}
  16 \{ p_1'' \wedge \text{arem}(ADD) \wedge (\text{MIN} < e < \text{MAX}) \wedge (e = E) \}
  17 \text{c} := \text{p.next};
  18 \{ p_1''(c) \wedge \text{arem}(ADD) \wedge (\text{MIN} < e < \text{MAX}) \wedge (e = E) \}
  19 \}
  20 \text{...}
\[ p_2 \text{ def } \exists z, A_1, u, A_2, L_1, s, L_2. \text{toaddcid}(\text{true}) \land \text{ls_irr}_{x_1,d_1}(\text{Head, A}_1, p) \land L_{cid}(p, v, c) \land N_{x_1,d_1,s}(c, u, z) \land \text{ls_irr}_{x_1,d_2}(z, A_2, \text{null}) \\
\land s(A_1::u::A_2) \land (1 + \text{len}(A_2) + |\text{toaddThrds}(|\text{tidset}(s::L_2)|)| \leq \text{aux}) \land (v < e < \text{MAX}) \]

\[ p''_{2} \text{ def } \exists z, A_1, u, A_2, L_1, s, L_2. \text{toaddcid}(\text{true}) \land \text{ls_irr}_{x_1,d_1}(\text{Head, A}_1, p) \land L_{cid}(p, v, c) \land \text{ls_irr}_{x_1,d_2}(z, A_2, \text{null}) \\
\land s(A_1::u::A_2) \land (1 + \text{len}(A_2) + |\text{toaddThrds}(|\text{tidset}(L_2)|)| \leq \text{aux}) \land (u < e < \text{MAX}) \]

\[ p''_{2}(c, z) \text{ def } \exists A_1, A_2, L_1, L_2. \text{toaddcid}(\text{true}) \land \text{ls_irr}_{x_1,d_1}(\text{Head, A}_1, c) \land L_{cid}(c, u, z) \land \text{ls_irr}_{x_1,d_2}(z, A_2, \text{null}) \\
\land s(A_1::u::A_2) \land (1 + \text{len}(A_2) + |\text{toaddThrds}(|\text{tidset}(L_2)|)| \leq \text{aux}) \land (u < e < \text{MAX}) \]

\[ \{ p''_{2}(c) \land \text{arem}(\text{ADD}) \land (\text{MIN} < e < \text{MAX}) \land (e = E) \} \]

\[ \langle u := c.\text{data}; \text{aux} := \text{metricTAS}(p)\rangle; \]

\[ \{ p_2 \land \text{\circ}(\text{aux}) \land \text{arem}(\text{ADD}) \land (e = E) \} \]

\[ \text{while} \ (u < e) \{ \]

\[ \{ p_2 \land \text{\circ}(\text{aux} - 1) \land (u < e) \land \text{arem}(\text{ADD}) \land (e = E) \} \]

\[ b := \text{false}; \]

\[ \{ p''_{2}(\text{b} \lor p_2 \land (u < e) \land \text{\neg}\text{b} \land \text{\circ}(\text{aux} - 1) \land \text{arem}(\text{ADD}) \land (e = E) \} \]

\[ \text{while} \ (\text{\neg}\text{b}) \{ \]

\[ b := \text{cas(c.lock, 0, cid)}; \]

\[ \} \]

\[ \{ p''_{2} \land \text{\circ}(\text{aux} - 1) \land \text{arem}(\text{ADD}) \land (e = E) \} \]

\[ p.\text{lock} := \text{0}; \]

\[ \{ \exists z. p''_{2}(c, z) \land \text{\circ}(\text{aux} - 1) \land \text{arem}(\text{ADD}) \land (e = E) \} \]

\[ p := c; \]

\[ c := p.\text{next}; \]

\[ \{ p''_{2}(p, c) \land \text{\circ}(\text{aux} - 1) \land \text{arem}(\text{ADD}) \land (e = E) \} \]

\[ \langle u := c.\text{data}; \text{aux} := \text{metricTAS}(p)\rangle; \]

\[ \{ p_2 \land \text{\circ}(\text{aux}) \land \text{arem}(\text{ADD}) \land (e = E) \} \]

\[ \} \]

\[ \{ p_2 \land (e \leq u) \land \text{arem}(\text{ADD}) \land (e = E) \} \]

\[ \} \]

\[ \ldots \]

---

**Figure 75.** Proof outline for add (2).

---

\[ p''_{2} \text{ def } \exists z, A_1, u, A_2, L_1, s, L_2. \text{toaddcid}(\text{true}) \land \text{ls_irr}_{x_1,d_1}(\text{Head, A}_1, p) \land L_{cid}(p, v, c) \land U(c, u, z) \land \text{ls_irr}_{x_1,d_2}(z, A_2, \text{null}) \\
\land s(A_1::u::A_2) \land (1 + \text{len}(A_2) + |\text{toaddThrds}(|\text{tidset}(s::L_2)|)| \leq \text{aux}) \land (u < e < \text{MAX}) \]

\[ p''_{2} \text{ def } \exists z, A_1, u, A_2, L_1, s, L_2. \text{toaddcid}(\text{true}) \land \text{ls_irr}_{x_1,d_1}(\text{Head, A}_1, p) \land L_{cid}(p, v, c) \land \text{ls_irr}_{x_1,d_2}(z, A_2, \text{null}) \\
\land s(A_1::u::A_2) \land (1 + \text{len}(A_2) + |\text{toaddThrds}(|\text{tidset}(s::L_2)|)| \leq \text{aux}) \land (u < e < \text{MAX}) \]

\[ \{ (p''_{2} \land b) \lor (p_2 \land (u < e) \land \text{\neg}\text{b}) \} \]

\[ \{ (p''_{2} \land b) \lor (p_2 \land (u < e) \land \text{\neg}\text{b} \land \text{\circ}(1)) \} \]

\[ \text{while} \ (\text{\neg}\text{b}) \{ \]

\[ \{ (p''_{2} \land \text{\circ}(0)) \lor (p''_{2} \land \text{\circ}(1)) \} \]

\[ b := \text{cas(c.lock, 0, cid)}; \]

\[ \{ (p''_{2} \land b) \lor (p_2 \land (u < e) \land \text{\neg}\text{b} \land \text{\circ}(1)) \} \]

\[ \} \]

\[ p''_{2} \]

---

**Figure 76.** Proof outline for add (2) – the loop at lines 11–13.
\[
\{ p_2 \land (e \leq u) \land \text{arem}(ADD) \land (e = E) \}
\]

\[
\exists z, A_1, v, A_2, L_1, s, L_2. \text{toaddcld(true)} \land \text{lsjrcld}_{L_2}((\text{Head}, A_1, p) + L_{cld}(p, v, c) \land N_{jrcld,s}(c, u, z))
\]

\[
\{ \text{lsjrcld}_{L_2}(z, A_2, nul) \land \text{ss}(A_1 : v : u : A_2) \land (v < e \leq u) \land (e < \text{MAX}) \land \text{arem}(ADD) \land (e = E) \}
\]

19 if (u != e) {
\[
\{ \exists z, A_1, v, A_2, L_1, s, L_2. \text{toaddcld(true)} \land \text{lsjrcld}_{L_2}((\text{Head}, A_1, p) + L_{cld}(p, v, c) \land N_{jrcld,s}(c, u, z))
\]

\[
\{ \text{lsjrcld}_{L_2}(z, A_2, nul) \land \text{ss}(A_1 : v : u : A_2) \land (v < e \leq u) \land \text{arem}(ADD) \land (e = E) \}
\]

20 \begin{align*}
& x := \text{cons}(0, e, c); \\
& \{ \exists z, A_1, v, A_2, L_1, s, L_2. \text{toaddcld(true)} \land \text{lsjrcld}_{L_2}((\text{Head}, A_1, p) + L_{cld}(p, v, c) \land U(x, e, c) \land N_{jrcld,s}(c, u, z)) \\
& \{ \text{lsjrcld}_{L_2}(z, A_2, nul) \land \text{ss}(A_1 : v : u : A_2) \land (v < e \leq u) \land \text{arem}(ADD) \land (e = E) \}
\end{align*}

21 \begin{align*}
& \langle p._\text{next} := x; \text{toaddcld} := \text{false}; \rangle \\
& \{ \exists z, A_1, v, A_2, L_1, s, L_2. \text{toaddcld(false)} \land \text{lsjrcld}_{L_2}((\text{Head}, A_1, p) + L_{cld}(p, v, x) \land U(x, e, c) \land N_{jrcld,s}(c, u, z)) \\
& \{ \text{lsjrcld}_{L_2}(z, A_2, nul) \land \text{ss}(A_1 : v : e : u : A_2) \land \text{arem}(<\text{skip}) \land (R = true) \\
& \}
\end{align*}

22 \begin{align*}
& r := \text{true}; \\
& \{ r := false; \text{toaddcld} := \text{false}; \}
\end{align*}

23 \} else {
24 \begin{align*}
& \text{toaddcld} := \text{false}; \\
& \{ \exists z, A_1, v, A_2, L_1, s, L_2. \text{toaddcld(false)} \land \text{lsjrcld}_{L_2}((\text{Head}, A_1, p) + L_{cld}(p, v, x)) \\
& \{ \text{lsjrcld}_{L_2}(z, A_2, nul) \land \text{ss}(A_1 : v : u : A_2) \land \text{arem}(<\text{skip}) \land (r = R) \\
& \}
\end{align*}

25 \begin{align*}
& \text{p.lock} := 0; \\
& \{ \exists A, \text{toaddcld(false)} \land \text{lsjrcld}_{L_2}((\text{Head}, A, nul) \land \text{ss}(A) \land \text{arem}(<\text{skip}) \land (r = R) \\
& \}
\end{align*}

26 \begin{align*}
& \{ P \land \text{arem}(<\text{skip}) \land (r = R) \}
\end{align*}

27 return r;
\}
\[ p_4 \overset{\text{def}}{=} \exists z, A_1, v, A_2, L_1, L_2. \text{toadd}(false) * \text{ls}_{cid}(Head, A_1, p) * L_{cid}(p, v, c) * U(c, e, z) * \text{ls}_{cid}(z, A_2, null) * \text{ss}(A_1 :: v :: e :: A_2) \]

\[ p_4' \overset{\text{def}}{=} \exists z, A_1, v, A_2, L_1, s, L_2. \text{toadd}(false) * \text{ls}_{cid}(Head, A_1, p) * L_{cid}(p, v, c) * \text{ls}_{cid}(s, c, e, z) * \text{ls}_{cid}(z, A_2, null) * \text{ss}(A_1 :: v :: e :: A_2) \]

\[ p_4'' \overset{\text{def}}{=} \exists z, A_1, v, A_2, L_1, L_2. \text{toadd}(false) * \text{ls}_{cid}(Head, A_1, p) * L_{cid}(p, v, c) * L_{cid}(c, e, z) * \text{ls}_{cid}(z, A_2, null) * \text{ss}(A_1 :: v :: e :: A_2) \]

\[ \text{rmv}(e) \{ \]
\n1. local p, c, n, pi, po, ci, co, u, r, aux;
\n\{ P \land \phi(1) \land (\text{MIN} < e < \text{MAX}) \land \text{arem} \land \text{rmv} \land (e = E) \} \]

2. p := Head;
...  
8. \langle u = c.data; aux := \text{metricTAS}(p) \rangle;
9. while (u < e) { 
...  
18. \} \}
19. if (u = e) { 
19. \{ \}
20. b := false;
21. \{ ((p_4'' \land b) \lor (p_4 \lor p_4') \land \neg b \land \phi(1)) \land (e < \text{MAX}) \land \text{arem} \land (e = E) \} \}
22. \}
23. \}
\}
24. n := c.next;
\{ \}
25. p.next := n;
\{ \}
26. p.lock := 0;
\{ \}
27. dispose(c);
\{ \}
28. r := true;
29. \} else { 
30. \}
31. p.lock := 0;
\{ \}
32. r := false;
\} \}
33. return r; \}

Figure 78. Proof outline for \text{rmv}.  

106
D.4 Optimistic list with TAS lock

Fig. 79 shows the code of the optimistic list implemented with TAS locks (where the auxiliary code is in red). In the optimistic list, a thread traverses the list without taking any locks, and when finding the candidate nodes, it locks the nodes and validates that they are still in the list and adjacent. If the validation fails, the nodes are unlocked and the operation is restarted. The code we show here is an optimized version of the code we discussed in Sec. 7. In Fig. 79, the thread locks only one node \( p \) before validation. It does not lock \( c \) until the thread is going to remove \( c \) at line 35. We prove it is deadlock-free.

Fig. 80 defines the precise invariant and the precondition. Fig. 81 defines the rely/guarantee conditions and the definite actions. Other figures in this section give the proof outlines.

```c
intSet gn; // initially empty
struct Node {
    int lock;
    int data;
    struct Node *next;
}
struct List {
    struct Node *Head;
}
initialize() {
    Head := cons(0, MIN, null);
    Head.next := cons(0, MAX, null);
}
add(e) {
    local p, c, x, s, done, b, w, v, u, r;
    done := false;
    while (!done) {
        p := Head;
        c := p.next;
        u := c.data;
        while (u < e) {
            p := c;
            c := c.next;
            u := c.data;
        }
        b := false;
        while (!b) {
            b := cas(p.lock, 0, cid);
        }
        v := p.data;
        s := Head;
        w := s.data;
        while (w < v) {
            s := s.next;
            w := s.data;
        }
        if (s = p && p.next = c) {
            done := true;
        } else {
            p.lock := 0;
        }
    }
    if (u != e) {
        x := cons(0, e, c);
        p.next := x;
        r := true;
    } else {
        r := false;
    }
    p.lock := 0;
    return r;
}
rmv(e) {
    local p, c, n, s, done, b, w, v, u, r;
    done := false;
    while (!done) {
        p := Head;
        c := p.next;
        u := c.data;
        while (u < e) {
            p := c;
            c := c.next;
            u := c.data;
        }
        b := false;
        while (!b) {
            b := cas(p.lock, 0, cid);
        }
        v := p.data;
        s := Head;
        w := s.data;
        while (w < v) {
            s := s.next;
            w := s.data;
        }
        if (s = p && p.next = c) {
            done := true;
        } else {
            p.lock := 0;
        }
    }
    if (u = e) {
        b := false;
        while (!b) {
            b := cas(c.lock, 0, cid);
        }
        n := c.next;
        c.lock := 0;
        r := true;
    } else {
        r := false;
    }
    p.lock := 0;
    return r;
}
```

Figure 79. Optimistic list with TAS lock.
As for the lock-coupling list, the invariant $I$ defined in Figure 80 requires the concrete list to be sorted and its elements to constitute the abstract set $S$. Since the optimistic algorithm ignores the locks when traversing the list, it may access nodes that have been removed from the list. Thus, we cannot dispose removed nodes as in the lock-coupling list. Instead, we need to introduce a write-only auxiliary variable $gn$ to remember those removed nodes (see line 35 in Fig. 79). The precise invariant $I$ should include those nodes (garb).

As shown in Figure 81, the delaying actions of a thread $t$ are stratified. The $Add$ and $Rmv$ actions which update the list are classified at level 2. The lock acquirements $Lock$ are at level 1. The $Unlock$ actions are not delaying actions.

The definite actions $D$ describe the various scenarios under which the lock release would definitely happen. Note that it also includes the lock release actions on removed nodes.

The $add$ method is given $\text{\bf \#}(1, 1)$ initially, where the 2-level $\text{\bf \#}$-token is for doing $Add$ and the 1-level $\text{\bf \#}$-token is for doing $Lock$. Fig. 82 and Fig. 83 show the proof outlines. Note that when the environment threads perform $Add$ or $Rmv$, the current thread could gain more 1-level $\text{\bf \#}$-tokens (by the stability checking) so that it can rollback and re-do its $Lock$ action. In the proofs, the assertions for inner loops are in cyan color, to be distinguished from the assertions for the outer loop. As we mentioned, we apply the (\text{\bf CONJ}) rule to reuse the proofs of the inner loop at the outer loop, which allows us to discard the tokens of the inner loop. Following earlier work [22, we also provide the (FR-CONJ) rule to add back the original tokens of the outer loop.

The $rmv$ method is given $\text{\bf \#}(1, 2)$ initially, where the 2-level $\text{\bf \#}$-token is for doing $Rmv$ and the two 1-level $\text{\bf \#}$-tokens are for doing the two $Lock$ actions respectively. The proof in Fig. 84 is similar to the proof of the add method.
\[ t \in def \equiv \bigvee_{\varphi \in T} G_{\varphi} \]

\[ G_{t} \equiv (Add_{t} \lor Rmv_{t} \lor Lock_{t} \lor Unlock_{t} \lor Id) \land (I \times I) \]

\[ Add_{t} \equiv \exists x, y, z, v, w, u, s, S. (Lt(x, v, z) \times (S = S)) \times_{2} (Lt(x, v, y) \times U(y, w, z) \times (S = S \cup \{w\})) \times [N_{irr_{s}}(z, u, n) \land (v < w < u)] \]

\[ Rmv_{t} \equiv \exists x, y, z, v, u, S_{y}. (Lt(x, v, y) \times (gn = S_{y}) \times (S = S \cup \{u\})) \times_{2} (Lt(x, v, z) \times (gn = S_{y} \cup \{y\}) \times (S = S)) \times [Lt(y, u, z) \land (u < \text{MAX})] \]

\[ Lock_{t} \equiv \exists x, y, v. Lt(x, v, y) \times_{1} Lt(x, v, y) \]

\[ Unlock_{t} \equiv \exists x, y, v. Lt(x, v, y) \times U(x, v, y) \]

\[ D_{c1a} \equiv (\forall x. dp_{c1a}(x) \leadsto dq_{c1a}(x)) \land (\forall x. dp_{c2a}(x) \leadsto dq_{c2a}(x)) \]

\[ dp_{c1a}(x) \equiv \exists y, z, A, v, u, A', L. lt_{irr_{L}}(\text{Head}, A, x) \times Lt(x, v, y) \times (U(y, u, z) \lor Lt(y, u, z)) \times \text{ls}\_unlocked(z, A', \text{null}) \times ss(A::v::u::A') \times \text{garb} \]

\[ dq_{c1a}(x) \equiv \exists y, z, A, v, u, A', L. lt_{irr_{L}}(\text{Head}, A, x) \times U(x, v, y) \times \text{ls}\_unlocked(y, A', \text{null}) \times ss(A::v::A') \times \text{garb} \]

\[ dp_{c2a}(x) \equiv \exists A, L, S_{y}. lt_{L}(\text{Head}, A, \text{null}) \times ss(A) \times (gn = S_{y} \cup \{x\}) \times (\forall y \in S_{y} N(y, \_\_), \text{null}) \times Lt(x, \_\_, \_\_) \]

\[ dq_{c2a}(x) \equiv \exists A, L, S_{y}. lt_{L}(\text{Head}, A, \text{null}) \times ss(A) \times (gn = S_{y} \cup \{x\}) \times (\forall y \in S_{y} N(y, \_\_, \_\,_)) \times U(x, \_\_, \_\_) \]

**Figure 81.** Rely, guarantee and definite actions of the optimistic list with TAS lock.
findLocked(x, v, u, y) ≡
∃z, A, A', s, L. ls_irr_cid.d((Head, A, x) * L_cid(x, v, y) * N.irr_cid.s(y, u, z) * ls_irr_cid.d(z, A', null) * ss(A::v::u::A') * garb

p1(v, A2, s1) ≡
∃z, A1, l1, s, l2, ls_irr_cid.d((Head, A1, p) * N.s1(p, v, c) * N.irr_cid.s(c, u, z) * ls_irr_cid.d2(z, A2, null) * ss(A1::v::u::A2) * garb

p1'(v, A2, s1) ≡
∃z, v, s, l2, N.s1(p, v, z) * N.irr_cid.s(c, u, z) * ls_irr_cid.d2(z, A2, null) * true
∧ (p ∈ gn ∨ v ∈ gn ∨ exists(x'::A', c) * true)

p2(v) ≡ ∃A2, p2(v, A2) p2(v, A2) ≡ ∃s1, P * p1(v, A2, s1) ∧ (s1 ≠ cid) p3(v) ≡ ∃A2, P' * p1(v, A2, cid)

p2'(v) ≡ ∃A2, p2'(v, A2) p2'(v, A2) ≡ ∃s1, P * p1'(v, A2, s1) ∧ (s1 ≠ cid) p3'(v) ≡ ∃A2, P' * p1'(v, A2, cid)

p4(A2) ≡ ∃z, s, l1, s, l2, ls1((Head, A1, s) * N.s(u, z) * ls2(z, A2, null) * ss(A1::v::u::A2) * garb

p4'(A2) ≡ ∃z, s, l2, N_worker.s1(s, v) * ls2(z, A2, null) * true ∧ (s ∈ gn)

add(e) {
  1 local p, c, s, done, b, v, u, r;
  2 done := false;
  3 while (!done) {
    4 done := true;
    5 c := p.next;
    6 u := c.data;
    7 { done ∧ ∃A2. (p2(v, A2) ∧ ì(1,1) ∨ 0(1) ∨ 0(1)) ∨ ∈(1,0) ∧ 0(1) ∧ 0(1) ∧ 0(1) ∧ 0(1)) ∧ (v < e ≤ MAX) ∧ arem(ADD) ∧ (e = E) }
    8 p := c;
    9 c := c.next;
  10 u := c.data;
  11 { done ∧ ∃A2. (p2(v, A2) ∧ ì(1,1) ∨ 0(1) ∧ 0(1) ∧ 0(1) ∧ 0(1) ∧ 0(1)) ∧ (v < e ≤ MAX) ∧ arem(ADD) ∧ (e = E) }
  12 b := false;
  13 done := true;
  14 { done ∧ ∃v. (b ∧ (p3(v) ∧ ì(1,0) ∨ 0(1) ∧ 0(1) ∧ 0(1)) ∨ ∈(1,0) ∧ 0(1) ∧ 0(1)) } ∨ 0(1) ∧ 0(1) ∧ 0(1)) } 
  15 while (b) {
  16 b := cas(p.lock, 0, cid);
  17 p := p.data;
  18 s := Head;
  19 w := s.data;
  20 { done ∧ ∃A2. (p2(v, A2) ∧ ì(1,1) ∨ 0(1) ∧ 0(1) ∧ 0(1)) ∧ (v < e ≤ u) ∧ arem(ADD) ∧ (e = E) }
  21 } (p1(v) ∧ ì(1,1) ∧ 0(1) ∧ 0(1) ∧ 0(1)) ∧ (v < e ≤ u) ∧ arem(ADD) ∧ (e = E) }
  22 while (w) {
  23 s := s.next;
  24 w := s.data;
  25 { done ∧ ∃A2. (p2(v, A2) ∧ ì(1,1) ∨ 0(1) ∧ 0(1) ∧ 0(1)) ∧ (v < e ≤ u) ∧ arem(ADD) ∧ (e = E) }
  26 if (s = p && p.next = c) {
  27 done := true;
  28 done ∧ findLocked(p, v, c, u) ∧ ì(1,0) ∧ (v < e ≤ u) ∧ arem(ADD) ∧ (e = E) }
  29 } else {
  30 p.lock := 0;
  31 done := true;
  32 done ∧ findLocked(p, v, c, u) ∧ ì(1,0) ∧ (v < e ≤ u) ∧ arem(ADD) ∧ (e = E) }
  33 ...
\[
p_5 \equiv \exists v, x. \text{findLocked}(p, v, x)
\]

\[
\{ \exists v. \text{findLocked}(p, v, c, u) \land \Diamond(1, 0) \land (v < u) \land \text{arem}(\text{ADD}) \land (e = E) \}
\]

\[
\begin{align*}
29 & \text{if (} u \neq e \text{)} \{} \\
& \{ \exists v. \text{findLocked}(p, v, c, u) \land \Diamond(1, 0) \land (v < u) \land \text{arem}(\text{ADD}) \land (e = E) \} \\
30 & x := \text{cons}(0, e, c); \\
31 & p.\text{next} := x; \\
32 & r := true; \\
& \{ p_5 \land \text{arem(skip)} \land (r = R) \} \\
33 & \} \text{ else } \{} \\
& \{ \exists v. \text{findLocked}(p, v, c, u) \land \Diamond(1, 0) \land (e = u) \land \text{arem}(\text{ADD}) \land (e = E) \} \\
34 & r := false; \\
& \{ p_5 \land \text{arem(skip)} \land (r = R) \} \\
35 & \} \\
36 & p.\text{lock} := 0; \\
& \{ P \land \text{arem(skip)} \land (r = R) \} \\
37 & \text{return } r;
\end{align*}
\]

**Figure 83.** Proof outline for `add (2)`.

\[
\text{adjLocked}(x, v, y, u, z) \equiv
\exists A, A', L. \text{ls_irrecl}_{c, L}(\text{Head}, A, x) \ast L_{\text{cid}}(x, v, y) \ast L_{\text{cid}}(y, u, z) \ast \text{ls_irrecl}_{c, L}(z, A', \text{null}) \ast \text{ss}(A :: u :: A') \ast \text{garb}
\]

\[
\text{rmv}(e) \{ \\
1 & \text{local p, c, n, s, done, b, v, u, r; } \\
& \{ P \land \Diamond(1, 2) \land \text{arem}(\text{RMV}) \land (\text{MIN} < e < \text{MAX}) \land (e = E) \} \\
2 & \text{done := false; } \\
3 & \text{while (!done)} \{} \\
& \ldots \ldots \\
28 & \{ \exists v. \text{findLocked}(p, v, c, u) \land \Diamond(1, 1) \land (v < u < \text{MAX}) \land \text{arem}(\text{RMV}) \land (e = E) \} \\
29 & \text{if (} u \neq e \text{)} \{} \\
& \{ \text{findLocked}(p, v, c, e) \land \Diamond(1, 1) \land \text{arem}(\text{RMV}) \land (e = E < \text{MAX}) \} \\
30 & b := false; \\
31 & \text{while (} b \text{)} \{} \\
32 & b := \text{cas(c.\text{lock}, 0, cid); } \\
33 & \{ \exists v. \text{adjLocked}(p, v, c, e, z) \land \Diamond(1, 0) \land \text{arem}(\text{RMV}) \land (e = E < \text{MAX}) \} \\
34 & n := c.\text{next; } \\
& \{ \text{adjLocked}(p, v, c, e, n) \land \Diamond(1, 0) \land \text{arem}(\text{RMV}) \land (e = E < \text{MAX}) \} \\
35 & \text{p.\text{next} := n; } gn := gn \cup \{c\}; \\
& \{ p_5 \land \text{arem(skip)} \land (R = \text{true}) \} \\
36 & c.\text{lock} := 0; \\
37 & r := true; \\
& \{ p_5 \land \text{arem(skip)} \land (r = R) \} \\
38 & \} \text{ else } \{} \\
& \{ \exists v. \text{findLocked}(p, v, c, u) \land \Diamond(1, 1) \land (v < u) \land \text{arem}(\text{RMV}) \land (e = E) \} \\
39 & r := false; \\
& \{ p_5 \land \text{arem(skip)} \land (r = R) \} \\
40 & \} \\
41 & p.\text{lock} := 0; \\
& \{ P \land \text{arem(skip)} \land (r = R) \} \\
42 & \text{return } r;
\}
\]

**Figure 84.** Proof outline for `rmv`.
D.5 Lazy list with TAS lock

Fig. 85 shows the code of the lazy list implemented with TAS locks (where the auxiliary code is in red). We prove it is deadlock-free.

Similar to the optimistic list, in lazy list, a thread traverses the list without taking any locks, and when finding the candidate nodes, it locks the nodes and validates that they are still in the list and adjacent. But now every node in the concrete list has an additional mark field. The `rmv(e)` method first logically removes the node by setting its mark field before the physical removal (unlinking it from the list). Since we mainly focus on progress properties, here we only show the proofs for the add and `rmv` methods, which involve blocking operations (i.e., lock acquirements). The contain method of the lazy list does not acquire locks. It can be verified using earlier technique for linearizability verification (e.g., [21, 30]), and we omit its proofs here.

Fig. 86 defines the precise invariant and the precondition. Fig. 87 defines the rely/guarantee conditions and the definite actions. Other figures in this section give the proof outlines.

As for the optimistic list, we introduce a write-only auxiliary variable `gn` to remember the removed nodes. We also introduce the auxiliary variable `tmark` for each thread to indicate whether the thread has performed marking but has not perform the physical removal.

```c
bool[] tmark; //initially all false
intSet gn; //initially empty
struct Node {
    int lock;
    int data;
    struct Node *next;
    bool mark;
}
struct List {
    struct Node *Head;
}
initialize() { Head := cons(0, MIN, null, false); Head.next := cons(0, MAX, null, false); }
add(e) {
    local p, c, x, done, b, u, r;
    1 done := false;
    2 while (!done) {
        3 p := Head;
        4 c := p.next;
        5 u := c.data;
        6 while (u < e) {
            7 p := c;
            8 c := c.next;
            9 u := c.data;
        }
        10 b := false;
        11 while (!b) {
            12 b := cas(p.lock, 0, cid);
        }
        13 b := false;
        14 while (!b) {
            15 b := cas(c.lock, 0, cid);
        }
        16 if (!p.mark && !c.mark && p.next = c)
            17 done := true;
        else {
            18 p.lock := 0;
            19 c.lock := 0;
        }
    }
    20 c.lock := 0;
    21 if (u = e) {
        22 <c.mark := true; tmarkid := true>;
        23 p.lock := 0;
        24 c.lock := 0;
    }
    25 return r;
}
rmv(e) {
    local p, c, n, done, b, u, r;
    1 done := false;
    2 while (!done) {
        3 p := Head;
        4 c := p.next;
        5 u := c.data;
        6 while (u < e) {
            7 p := c;
            8 c := c.next;
            9 u := c.data;
        }
        10 b := false;
        11 while (!b) {
            12 b := cas(p.lock, 0, cid);
        }
        13 b := false;
        14 while (!b) {
            15 b := cas(c.lock, 0, cid);
        }
        16 if (!p.mark && !c.mark && p.next = c)
            17 done := true;
        else {
            18 p.lock := 0;
            19 c.lock := 0;
        }
    }
    20 if (u = e) {
        21 <c.mark := true; tmarkid := true>;
        22 n := c.next;
        23 p.next := n;
        24 <gn := gn \{c}; tmarkid := false>;
        25 r := true;
    }
    26 }
    27 if (u = e) {
        28 <c.mark := true; tmarkid := true>;
        29 n := c.next;
        30 p.next := n;
        31 r := true;
    }
    32 else {
        33 r := false;
    }
    34 p.lock := 0;
    35 c.lock := 0;
    36 return r;
}
```

Figure 85. Lazy list with TAS lock.
As shown in Figure 87, the delaying actions of a thread $t$ are stratified. The $Add$, $Rmv$ and $Mark$ actions which update the list are classified at level 2. The lock acquiring action $Lock11$ with certain constraints is at level 1. When a thread does $Lock11$, it must locks a node $x$ which is really on the list and other nodes behind $x$ (i.e., the nodes which are closer to the tail of the list) must be neither locked nor marked. $Lock21$ acquires the lock of the successor node with similar constraints. It is also at level 1. Other lock acquirments $Lock1$ and $Lock2$ are at level 0, which we do not viewed as delaying actions.

The add method is given $\ell(1, 2)$ initially, where the 2-level $\bullet$-token is for doing $Add$ and the two 1-level $\bullet$-tokens are for doing $Lock11$ and $Lock21$. Fig 88 and Fig 89 show the proof outlines. Note that the thread does not need to lose level-1 $\bullet$-tokens when the node it attempts to lock has been marked by an environment thread. It consumes level-1 $\bullet$-tokens for only $Lock11$ and $Lock21$ actions. Thus for $Lock11$ and $Lock2$ actions, the thread still keeps the level-1 $\bullet$-tokens so that it can roll back and do $Lock11$ and $Lock21$ in the future. Similar to the proofs for the optimistic list, the assertions for inner loops here are also in cyan color, to be distinguished from the assertions for the outer loop. We apply the (HIDE-$\cdot$) rule to reuse the proofs of the inner loop at the outer loop, which allows us to discard the tokens of the inner loop. Following earlier work [23], we also provide the (IR-CON) rule to add back the original tokens of the outer loop.

The rmv method is given $\ell(2, 2)$ initially, where the two 2-level $\bullet$-token are for doing $Mark$ and $Rmv$. The proof in Fig. 90 is similar to the proof of the add method.
\[ R_i \triangleq \bigvee_{t' \in G_i} G_{t'} \]

\[ G_1 \triangleq (Addi \lor Marki \lor Rmvi \lor Lock1i \lor Lock2i \lor LockI_1 \lor Unlock_1 \lor Id) \lor \text{Id} \land (I \times I) \]

\[ Addi \triangleq \exists x, y, z, n, v, w, u, s. ((L_i(x, y, z, false) \star (S = S)) \star 2 (L_i(x, y, y, false) \star U(y, w, z, false) \star (S = S \cup \{w\})) \]

\[ \star [N_{\text{marks}}(z, u, n, false) \land (w < w < u)] \]

\[ Marki \triangleq \exists x, y, z, u, s. ((L_i(x, y, y, false)) \star (L_i(y, u, z, false) \star (\text{tmark}_{k} = false) \star (S = S \cup \{u\}) \land (u < \text{MAX})) \]

\[ \star 2 (L_i(y, u, z, true) \star (\text{tmark}_{k} = true) \star (S = S)) \]

\[ Rmvi \triangleq \exists x, y, z, v, u, S_i. ((L_i(x, y, v, false)) \star (\text{tmark}_{k} = true) \star (\text{gn} = S_i)) \]

\[ \star (L_i(x, y, z, false) \star (\text{tmark}_{k} = false) \star (\text{gn} = S_i \cup \{y\})) \star [L_i(y, u, z, true) \land (u < \text{MAX})] \]

\[ Lock1i \triangleq \exists x, y, z, A, v, u. ([\text{is\_marks}(Head, A, x) \land (v < \text{MAX})] \star (U(x, y, v, false) \star 1 L_i(x, y, v, false)) \]

\[ \star [\text{is\_unlocked\_unmarked}(y, A_2, null)] \]

\[ Lock1_1 \triangleq \exists x, y, z, A_2, b, L_2, s, L_2'. (U(x, v, y, b) \times L_i(x, y, b)) \]

\[ \star [\text{is\_marks}(z, b', tmark) \star (\text{is\_marks}(z', b') \star (x, A_2, null) \times N_{\text{marks}}(z, z', b') \star (b = true \lor b' = true \lor s \neq 0)] \]

\[ Lock2_1 \triangleq \exists x, y, z, A_1, v, u, A_1, L_1, L_2, M_2. ([\text{is\_marks}(Head, A, x) \times L_i(x, y, v, false) \times (v < \text{MAX})] \times (U(y, u, z, false) \star 1 L_i(y, u, z, false)) \]

\[ \star [\text{is\_unlocked\_unmarked}(z, A_2, null, M_2) \times \{b \in M_2. b = false\}] \]

\[ Lock2_2 \triangleq \exists x, y, z, A, b, b_2, L_2, M_2. (L_i(x, v, y, b) \times \text{MAX}) \times (U(y', u, z, b_2) \times L_i(y', u, z, b_2)) \]

\[ \star [\text{is\_marks}(z, A, null, M_2) \star \text{true} \star (b_1 = true \lor b_2 = true \lor b_2 \in M_2 \lor \exists v', A', L_i(y', v', z, z, b') \star \text{true})] \]

\[ Unlock \triangleq \exists x, y, v, b. ([\text{tmark}_{s}(false) \star (L_i(x, y, v, b) \times U(x, v, y, b))] \]

\[ D_{\text{cid}} \triangleq (\forall x. dp_{\text{cid}}(x) \sim dq_{\text{cid}}(x)) \land (\forall x. dp'_{\text{cid}}(x) \sim dq'_{\text{cid}}(x)) \land (\forall x. dp''_{\text{cid}}(x) \sim dq''_{\text{cid}}(x)) \]

\[ dp_{\text{cid}}(x) \triangleq \exists y, z, A, v, u, A'. \text{tmark}_{s}(false) \star [\text{is\_mark}(Head, A, x) \star \text{L}_i(x, y, v, false) \star (U(y, u, z, false) \lor L_i(y, u, z, false)) \]

\[ \star [\text{is\_unlocked\_unmarked}(z, A', null) \star \text{ss}(A :: (v, false) :: (u, false) :: A') \star \text{garb}] \]

\[ dp_{\text{cid}}'(x) \triangleq \exists y, z, A, v, u, A'. \text{tmark}_{s}(false) \star [\text{is\_mark}(Head, A, x) \star \text{L}_i(x, y, v, false) \star (U(y, u, z, false) \lor L_i(y, u, z, false)) \]

\[ \star [\text{is\_unlocked\_unmarked}(z, A', null) \star \text{ss}(A :: (v, false) :: (u, false) :: A') \star \text{garb}] \]

\[ dp_{\text{cid}}''(x) \triangleq \exists y, z, A, v, u, A'. \text{tmark}_{s}(false) \star [\text{is\_mark}(Head, A, x) \star \text{L}_i(x, y, v, false) \star (U(y, u, z, false) \lor L_i(y, u, z, false)) \]

\[ \star [\text{is\_unlocked\_unmarked}(z, A', null) \star \text{ss}(A :: (v, false) :: (u, true) :: A') \star \text{garb}] \]

\[ dp_{\text{cid}}'''(x) \triangleq \exists y, z, A, v, u, A'. \text{tmark}_{s}(false) \star [\text{is\_mark}(Head, A, x) \star \text{L}_i(x, y, v, false) \star (U(y, u, z, false) \lor L_i(y, u, z, false)) \]

\[ \star [\text{is\_unlocked\_unmarked}(y, A', null) \star \text{ss}(A :: (v, false) :: A') \star \text{garb}] \]

\[ Figure \ 87. \ \text{Rely, guarantee and definite actions of the lazy list with TAS lock.} \]
add(e) {
  1  local p, c, x, done, b, u, r;
  2  { P ∧ ♦(1, 2) ∧ arem(ADD) ∧ (MIN < e < MAX) ∧ (e = E) }
  3  done := false;
  4  { ¬done ∧ P ∧ ♦(1, 2) ∧ ♦(1) ∨ ♦. done ∧ adjLocked(p, v, c, u, false, false, false) ∧ ♦(1, 0) ∧ u < e ≤ u) }
  5  { ¬arem(ADD) ∧ (MIN < e < MAX) ∧ (e = E) }
  6  while (!done) {
  7    p := Head;
  8    c := p.next;
  9    u := c.data;
  10    { ¬done ∧ ♦. (p2(v, A2) ∨ p′(v, A2) ∨ 0(1)) ∧ ♦(1, 2) ∧ (v < e < MAX) ∧ arem(ADD) ∧ (e = E) }
  11    { ¬done ∧ ♦. (p2(v) ∨ p′(v) ∨ 0(1)) ∧ ♦(1, 2) ∧ (v < e ≤ u) ∧ arem(ADD) ∧ (e = E) }
  12    b := false;
  13    { ¬done ∧ ♦. (b ∨ ♦(p2(v) ∨ p′(v) ∨ 0(1)) ∨ ♦. done ∧ adjLocked(p, v, c, u, false, false, false) ∧ ♦(1, 0) ∧ u < e ≤ u) }
  14    { ¬arem(ADD) ∧ (MIN < e < MAX) ∧ (e = E) }
  15    while (!b) {
  16      p.lock := 0;
  17      c.lock := 0;
  18      { ¬arem(ADD) ∧ (MIN < e < MAX) ∧ (e = E) }
  19    }  
  20    if (p.mark & & ♦. mark & & p.next = c) {
  21      done := true;
  22    } else {
  23      p.lock := 0;
  24      c.lock := 0;
  25      { ¬arem(ADD) ∧ (MIN < e < MAX) ∧ (e = E) }
  26    }
  27  }

Figure 88. Proof outline for add (1).
\begin{figure}
\begin{verbatim}
\textbf{p0} \text{def} = \\
\exists z, A, A', s, L \text{. tmarks}(false) \& L_{\text{cid}}(\text{Head, A, p}) \& L_{\text{cid}}(p, v, c, false) \& N_{\text{cid}, s}(c, u, z, false) \\
\& L_{\text{cid}}(z, A', \text{null}) \& ss(A :: (v, false) :: (u, false) :: A') \& \text{garb}
\end{verbatim}
\begin{verbatim}
\textbf{p0} \text{def} = \\
\exists z, A, v, A', L \text{. tmarks}(false) \& L_{\text{cid}}(\text{Head, A, p}) \& L_{\text{cid}}(p, v, z, false) \& L_{\text{cid}}(z, A', \text{null}) \& ss(A :: (v, false) :: A') \& \text{garb}
\end{verbatim}
\begin{verbatim}
{ \exists v. \text{adjLocked}(p, v, c, u, false, false) \& (1, 0) \& (\forall < e \leq u) \& \text{arem}(\text{ADD}) \& (e = E) }
27 c.lock := 0; \\
{ \exists v. p0(v) \& (1, 0) \& (\forall < e \leq u) \& \text{arem}(\text{ADD}) \& (e = E) }
28 if (u != e) { \\
{ \exists v. p0(v) \& (1, 0) \& (\forall < e < u) \& \text{arem}(\text{ADD}) \& (e = E) }
29 x := cons(0, e, c, false); \\
30 p.next := x; \\
31 r := true; \\
{ p0 \& \text{arem}(\text{skip}) \& (r = R) }
32 } else { \\
{ \exists v. p0(v) \& (1, 0) \& (\forall = u) \& \text{arem}(\text{ADD}) \& (e = E) }
33 r := false; \\
{ p0 \& \text{arem}(\text{skip}) \& (r = R) }
34 } \\
{ p0 \& \text{arem}(\text{skip}) \& (r = R) }
35 p.lock := 0; \\
{ P \& \text{arem}(\text{skip}) \& (r = R) }
36 return r;
}
\end{verbatim}
\caption{Proof outline for \text{add} (2).}
\end{figure}

\begin{figure}
\begin{verbatim}
rmv(e) {
1  local p, c, n, done, b, u, r;
{ P \& (2, 2) \& \text{arem}(\text{RMV}) \& (\forall < e < \text{MAX}) \& (e = E) }
2  done := false;
3  while (!done) {
.. . . .
26 } \\
{ \exists v. \text{adjLocked}(p, v, c, u, false, false) \& (2, 0) \& (\forall < e < \text{MAX}) \& \text{arem}(\text{RMV}) \& (e = E) }
27 if (u = e) { \\
{ \text{adjLocked}(p, c, e, false, false) \& (2, 0) \& \text{arem}(\text{RMV}) \& (e = E < \text{MAX}) }
28 <c.mark := true; \text{tmark} := \text{true}; \\
{ \text{adjLocked}(p, c, e, false, true, true) \& (1, 0) \& \text{arem}(\text{skip}) \& (e < \text{MAX}) \& (r = \text{true}) }
29 n := c.next; \\
<r.p.next := n; gn := gn \cup \{c\}; \text{tmark} := \text{false}; \\
r := true;
{ p0 \& \text{arem}(\text{skip}) \& (r = R) }
32 } else { \\
{ \exists v. \text{adjLocked}(p, v, c, u, false, true) \& (2, 0) \& (\forall < e < u) \& \text{arem}(\text{RMV}) \& (e = E) }
33 r := false; \\
{ p0 \& \text{arem}(\text{skip}) \& (r = R) }
34 } \\
{ p0 \& \text{arem}(\text{skip}) \& (r = R) }
35 p.lock := 0; \\
c.lock := 0; \\
{ P \& \text{arem}(\text{skip}) \& (r = R) }
37 return r;
}
\end{verbatim}
\caption{Proof outline for \text{rmv}.}
\end{figure}