A Program Logic for Contextual Refinement of Concurrent Objects under Fair Scheduling

Hongjin Liang  
Xinyu Feng  
University of Science and Technology of China

Abstract—Existing program logics on concurrent object verification either ignore progress properties, or aim for non-blocking progress (e.g., lock-freedom and wait-freedom), which cannot be applied to blocking algorithms that progress only under fair scheduling. In this paper we propose a new program logic for compositional verification of contextual refinement of concurrent objects under fair scheduling. As a key application, we show that starvation-freedom and linearizability of concurrent objects with blocking algorithms can be reformulated as contextual refinement, which can be verified using our program logic. With the logic, we have successfully verified starvation-freedom of simple algorithms using ticket locks, the two-lock queue algorithm and the lock-coupling list algorithm.

I. INTRODUCTION

A concurrent object or library provides a set of methods that allow multiple client threads to manipulate the shared data structure with abstract atomic behaviors. Blocking synchronization (i.e., mutual exclusion locks), as a straightforward technique to ensure exclusive accesses and control the interference, has been widely-used in object implementations to achieve linearizability, a classic correctness definition for concurrent objects [11].

In addition to linearizability, which is a safety property, object implementations are expected to also satisfy progress properties. The non-blocking progress properties, such as wait-freedom and lock-freedom, have been verified in recent work (e.g., [6], [12], [17]). They guarantee the termination of object methods independently of how the threads are scheduled.

Unfortunately these non-blocking progress properties are too strong to be satisfied by lock-based algorithms. 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 acquiring 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 [10], there are two desirable progress properties for blocking algorithms, deadlock-freedom and starvation-freedom. Both of them assume fair scheduling.

Recent program logics for verifying concurrent objects either prove only linearizability and ignore the issue of termination (e.g., [24], [3], [15], [23]), or aim for non-blocking progress properties (e.g., [6], [12], [17]), which cannot be applied to blocking algorithms that progress only under fair scheduling. One may think blocking algorithms are usually more coarse-grained than non-blocking ones, thus the verification problem should be much simpler. This might be true for safety properties, but the fairness assumption needed by the progress of blocking algorithms actually introduces complicated interdependencies among progress properties of threads, making it probably more challenging to verify the lock-based algorithms than their non-blocking counterparts.

For instance, starvation-freedom of concurrent objects requires that under fair scheduling a method call in any thread will finally return. For lock-based algorithms, it requires that every thread trying to acquire a lock should eventually succeed. The thread requesting the lock may wait for some environment threads to successfully acquire the same lock and eventually release it. Every progress of those environment threads can make the current thread closer to its own progress. The fair scheduling ensures that these environment threads eventually get scheduled to give their helps, which are crucial for the progress of the current thread. Therefore one of the key challenges we need to address is to take advantage of the fairness assumption, a non-local liveness property, in the Hoare-style compositional reasoning. We will explain the challenges in detail in Sec. II.

In this paper we propose a new program logic for compositional verification of contextual refinement of concurrent objects under fair scheduling. Recent work has shown the equivalence between contextual refinement and linearizability [5], therefore one can verify linearizability through refinement verification [3], [15]. Our new program logic supports verification of general termination-preserving refinement under fair scheduling. When the specification is instantiated to an abstract atomic operation, our logic not only establishes linearizability of the concrete implementation, but also ensures starvation-freedom. Our work is based on earlier work on concurrency verification, but we make the following new contributions:

- We propose a two-phase reasoning mechanism to establish progress under fair scheduling by induction over the number of the environment threads being waited by the current thread. We introduce a novel notion called a “definite action”, which models behaviors that can be definitely completed by a thread without relying on the progress of its environment. In the first phase (the base case), we establish the definite action; and in the second phase (the inductive step), we prove that a thread waiting for \( n + 1 \) environment threads can progress if these environment threads all have the definite action.
- We formally prove that, when the abstract specification used in the refinement proof is instantiated to an atomic...
Finally we discuss the related work in Sec. VII. We show the relationship to linearizability in Sec. IV. We present our new program and explain our approach informally in Sec. II. We define the correctness, which requires the concrete implementation of an object operation to have the same effect with its abstract counterpart. Thus the whole operation, the contextual refinement under fair scheduling ensured by the logic is equivalent to linearizability and starvation-freedom together. Therefore our logic can also be viewed as the first program logic for verifying linearizable and starvation-free concurrent objects.

- We apply our logic to verify simple objects with coarse-grained synchronization using ticket lock [19] and various queue locks (including Anderson array-based lock, CLH lock and MCS lock) [9]. For examples with more permissive locking scheme, we successfully verify the two-lock queue algorithm [20] and the lock-coupling list algorithm [9]. To the best of our knowledge, we are the first to verify starvation-freedom of lock-coupling list.

In the rest of this paper, we first analyze the challenges and explain our approach informally in Sec. II. We define the contextual refinement in Sec. III and present our new program logic in Sec. IV. We show the relationship to linearizability and starvation-freedom in Sec. V and the examples in Sec. VI. Finally we discuss the related work in Sec. VII.

II. INFORMAL DEVELOPMENT

A. Verifying contextual refinement for functional correctness

We start from a simple logic for verifying functional correctness, which requires the concrete implementation $C$ of an object operation to have the same effect with its abstract specification $\mathcal{C}$. This correctness criterion can be formulated as a contextual refinement between $C$ and $\mathcal{C}$. We say $C$ is a contextual refinement of $\mathcal{C}$, written as $C \sqsubseteq \mathcal{C}$, if substituting $C$ for $\mathcal{C}$ in any context (i.e., in a client program) does not add observable behaviors. For functional correctness, the observable behaviors are finite traces of external events such as I/O operations and runtime errors. Recent work (e.g., [5], [15]) has proved the equivalence between such a contextual refinement and linearizability [11], another commonly-accepted safety criterion for concurrent objects, for the special case when the specification $\mathcal{C}$ is an instantaneous atomic operation.

Fig. 1(b) shows a counter object implemented using a compare-and-swap (cas) instruction. The counter $x$ is atomically updated when the cas succeeds, producing the same effect with the abstract atomic operation INC in Fig. 1(a).

To prove $\lfinc \sqsubseteq \mathcal{INC}$, an intuitive approach is to find a linearization point (LP) in the code of the implementation $\lfinc$, and show it fulfills the abstract operation INC. The LP of $\lfinc$ is at the cas instruction when it succeeds (line 4). We maintain the consistency between the states at the concrete and the abstract sides by executing INC simultaneously with the concrete step at LP. Note that we may need to find multiple program points in $C$ to execute the abstract operation $\mathcal{C}$ when $C$ is not atomic (e.g., see the synchronous queue algorithm [9] verified in our previous work [17]).

We can encode the above LP-based reasoning in existing concurrent program logics, such as Rely-Guarantee reasoning and CSL. Following our earlier work [17], we introduce a new assertion $\text{arem}(\mathcal{C})$ to specify as auxiliary state the abstract operation $\mathcal{C}$ remaining to be fulfilled, which should be logically executed at LP of the concrete implementation. For the example in Fig. 1(b), we prove a judgment in the form of $I \vdash \{P \land \text{arem}(\mathcal{INC})\} \lfinc \{Q \land \text{arem}(\text{skip})\}$. Here $I$ denotes the interference with the environment. $P$ and $Q$ are relational assertions specifying the consistency relation between the program states at the concrete and the abstract sides. The postcondition in the above judgment shows that at the end of $\lfinc$ there is no remaining abstract operation to fulfill. At line 4 of $\lfinc$ we need to prove the following:

$$\{\ldots \land \text{arem}(\mathcal{INC})\}$$
$$\text{done} := \text{cas}(\&x, t, t+1);$$
$$\{\ldots \land \text{arem}(\text{skip}) \lor \neg \text{done} \land \text{arem}(\mathcal{INC})\}$$

That is, the abstract operation INC is fulfilled at the time when the cas succeeds.

The contextual refinement $C \sqsubseteq \mathcal{C}$ characterizes the functional correctness of the objects. But it does not characterize the progress properties. Consider the following example.

nop: while(true) skip;  NOP: skip;

We can prove that $\text{nop} \sqsubseteq \text{NOP}$ holds ($\text{nop}$ is linearizable with respect to $\text{NOP}$), since the definition of $C \sqsubseteq \mathcal{C}$ treats only finite event traces as observable behaviors. But $\text{nop}$ does not preserve the termination of $\text{NOP}$.

B. Preserving the termination under fair scheduling

We extend the contextual refinement $C \sqsubseteq \mathcal{C}$ to ensure the preservation of termination. A natural approach is to observe divergence (i.e., non-termination) over complete executions. We write $C \sqsubseteq^\omega \mathcal{C}$ as a contextual refinement requiring every client program which diverges using the concrete object $C$ to also diverge with the abstract operations $\mathcal{C}$. It is clear that $\text{nop} \sqsubseteq^\omega \text{NOP}$ does not hold for the above example.

For the implementation $\lfinc$ in Fig. 1(b), we have $\lfinc \sqsubseteq^\omega \mathcal{INC}$. In fact, $\lfinc$ is lock-free, ensuring that some method call will finish in a finite number of steps [9]. Whenever client threads execute $\lfinc$ concurrently, there always exists some successful update of $x$. Thus the whole client program calling $\lfinc$ diverges only if there are an infinite number of method calls. Then this client must also diverge when using the abstract operation INC.

The contextual refinement $C \sqsubseteq^\omega \mathcal{C}$ characterizes the full correctness of concurrent objects, but it does not apply to a real-world setting where the scheduling is fair. We write
$C \sqsubseteq^C C$ for the contextual refinement under fair scheduling, which cares about only fair executions at the concrete and the abstract sides. $C \sqsubseteq^\omega C$ does not imply $C \sqsubseteq^C C$, because the former allows some observable behaviors generated by a fair execution at the concrete side to be produced by an unfair execution at the abstract side. Consider the client $cl$ below.

\[
\begin{align*}
\text{lfInc; print(1); while (true) {lfInc; }} \\
\text{INC; print(1); while (true) { INC; }}
\end{align*}
\]

It may never print out 1 under fair scheduling. The case instruction in $lfInc$ called by the left thread may continuously fail due to the updates of $x$ made by the right thread. But for the client $CL$ using the atomic operation $INC$, as shown below, it will always print out 1 under fair scheduling.

\[
\begin{align*}
\text{INC; print(1); while (true) { INC; }}
\end{align*}
\]

Thus $lfInc \sqsubseteq^F INC$ does not hold. The reason we still have $lfInc \sqsubseteq^\omega INC$ is that it allows unfair scheduling at the abstract side, so the left thread in $CL$ may not have a chance to be scheduled, which corresponds to the case that the left thread diverges in $cl$.

To prove termination-preservation in either $C \sqsubseteq^F C$ or $C \sqsubseteq^\omega C$, the key challenge is to prevent $C$ from running forever without actually refining any step of $C$ (otherwise we may allow a diverging $C$ to refine a terminating $C$). In our earlier work [17] for proving $C \sqsubseteq^\omega C$, we introduce a counter $n$ (i.e., the number of tokens assigned to the current thread), and require that the thread can only run the loop for no more than $n$ rounds before it fulfills one or more abstract moves. The counter is treated as an auxiliary state, and decreases at the beginning of every round of loop (i.e., we pay one token for each iteration).

In a general setting, $C$ does not have to be atomic. It may execute multiple steps or even diverge. In this case, we allow $C$ to loop more when some abstract operations are fulfilled, by resetting the counter (i.e., increasing the number of tokens). The rule for loops is in the following form:

\[
\frac{P \land B \Rightarrow P' \ast \text{wf}(1) \quad I \vdash \{P'\}C(P)}{I \vdash \{P\} \text{while } (B) \ C\{P \land \neg B\}\text{ (LOOP)}}
\]

Here $\text{wf}(1)$ represents one token, and “$\ast$” 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. During the execution of $C$, the number of tokens could be increased if some abstract steps take place.

In this paper, we follow the similar token-based idea to prove termination-preservation in $C \sqsubseteq^F C$, but things become more complex here due to the assumption of fair scheduling. First, the progress of one thread may be dependent upon another. We have to be careful to avoid reasoning relying on circular progress dependency, a well-known problem that may lead to unsound logic. Second, we need to find a way to take advantage of the fairness assumption, a non-local liveness property, in our program logic for local reasoning. We address these problems by introducing the two-phase reasoning below.

### C. Two-phase reasoning

Fig. 1(c) shows $sfInc$, a counter object with coarse-grained synchronization using the ticket lock implementation [19]. It uses the shared variables $owner$ and $next$ to guarantee the first-come-first-served property of the lock. Initially $owner$ equals $next$. To acquire the lock, a thread atomically increments $next$ and reads its old value to a variable $i$ (line 2). The value of $i$ becomes the thread’s ticket. The thread waits for $owner$ to equal its ticket value $i$ (line 3). When the loop at line 3 terminates, the thread enters its critical section (line 4). The lock is released by incrementing $owner$ (line 5) such that the next waiting thread (the thread with ticket $i + 1$, if there is one) can now enter the critical section.

Note that the code in the critical section (line 4) could be non-atomic. Nevertheless, $sfInc$ is linearizable with respect to the atomic $INC$ in Fig. 1(a), i.e., $sfInc \subseteq INC$ holds, because $sfInc$ ensures mutually exclusive access to $x$. Any instruction after line 3 can be viewed as its LP.

Besides, $sfInc$ is starvation-free. Intuitively the ticket lock provides the first-come-first-served guarantee: if a thread $t_1$ requests the lock ahead of another thread $t_2$ (as a consequence, $t_1$’s ticket number must be less than $t_2$’s), then $t_1$ must acquire the lock ahead of $t_2$. Moreover, the critical section always terminates since it is straight-line code. Thus the threads which request/acquire the lock ahead of the current thread $t$ can eventually release the lock. Then we know $t$ must eventually acquire the lock and finish its operation. Thus for any client, substituting $sfInc$ for $INC$ in fair executions does not add observable behaviors (including termination/divergence behaviors). That is, $sfInc \sqsubseteq^F INC$ holds.

It is difficult to encode the above reasoning in a Hoare-style thread-local proof, because the progress of the current thread $t$ relies on the progress of its environment. To prove that $t$ can succeed in acquiring the lock, we need to show that the environment threads which request the lock ahead of $t$ can acquire the lock. Apparently we are trapped in circular reasoning, which is dangerous (usually unsound) for proving liveness properties. As an example, the following code may not terminate, although we can prove termination of each thread under the assumption of termination of the other.

\[
\begin{align*}
x &:= 1; \\
\text{while } (x = 1) \text{ skip; }
\end{align*}
\]

We resolve the apparent circularity with induction proofs. Our idea is based on the following observations over most starvation-free algorithms:

- There always exists a thread whose progress does not rely on its environment.
- For the thread $t$ whose progress relies on some environment threads, every progress of these environment threads helps the progress of $t$, i.e., makes $t$ get closer to a state at which $t$ can progress on its own.

In the example $sfInc$ in Fig. 1(c), the threads which are currently requesting the lock constitute a queue $t_1, t_2, \ldots, t_n$. The first thread, $t_1$, can immediately acquire the lock. Its progress
does not rely on its environment. For each thread \( t_k \) where \( k > 1 \), its progress relies on the progress of its environment threads \( t_1, \ldots, t_{k-1} \). When any of these environment threads releases the lock, \texttt{owner} is incremented and the thread \( t_k \) gets closer to a state where its ticket number equals \texttt{owner}. After all these environment threads progress, the thread \( t_k \) must end in a state where it can progress on its own.

Based on the above observations, we can verify the progress of a thread \( t \) by induction over “the amount of the helps” that it needs from its environment threads. Usually we can understand “the amount of the helps” as the number of the environment threads which need to help \( t \) progress. The case could be more complicated, e.g., in the proof for the lock release the lock, \( t \) must end in a state where its ticket number equals \texttt{owner}. After all these environment threads progress, the thread \( t_k \) must end in a state where it can progress on its own.

The rule for while loops is as follows.

\[
\frac{I, D ∪ \{P \land B\}C\{P\} \quad P \Rightarrow J \quad Stable(J) \quad \text{\texttt{J}@E} \, D \, \Rightarrow \, Q \quad Stable(Q)}{P \land B \land Q \Rightarrow \mathcal{P}' \ast \text{\texttt{wt}(1)} \quad I, D ∪ \{\mathcal{P}'\}C\{P\}}
\]

The rule needs us to first find two special assertions \( J \) and \( Q \). \( J \) is an invariant preserved at every step during the executions of the loop, which is required by \( P \Rightarrow J \) and \( Stable(J) \). \( Q \) specifies the states at which the current thread can progress on its own. Once \( Q \) holds, it should be preserved, as required by \( Stable(Q) \). The last two premises incorporate the token-based idea in the (LOOP) rule we just explained, which guarantees to preserve termination of the abstract code when \( Q \) holds.

The premise \( \text{\texttt{J}@E} \, D \, \Rightarrow \, Q \) is the key to the inductive step. Here \( E \) is a well-founded metric about “the amount of the helps” that the current thread needs from its environment for progress. \( \text{\texttt{J}@E} \, D \, \Rightarrow \, Q \) says that at any state satisfying \( J \), either \( Q \) holds (the current thread can progress on its own), or \( E \) decreases after the definite action \( D \) made by certain thread \( t' \) in the environment. It also requires \( E \geq 0 \) if \( J \) holds. Since \( J \) is an invariant, by induction over \( E \), we know that eventually \( Q \) holds, which implies the progress of the loop.

The second premise implicitly exploits the fairness assumption. The fair scheduling ensures that the aforementioned thread \( t' \) in the environment is scheduled infinitely often. Since we prove in phase 1 that the definite action is indeed definite without relying on the environment, we know the environment’s helps needed by the current thread eventually take place. We can reason about the progress of the thread regardless of the time when these helps take place because \( J \) is an invariant.

Note that not all the rules in this section are simplified versions to demonstrate the ideas. The actual rules are given in Sec. IV.

## III. Basic Technical Settings

### A. The programming language

In Fig. 2 we give the syntax of the language. A program \( W \) consists of multiple parallel threads sharing an object \( O \). 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 command \texttt{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. \texttt{noret} and \texttt{end} are commands that will be inserted into the program at runtime to define operational semantics, and are not supposed to be used directly by programmers. \texttt{noret} is appended at the end of object method body when the method is called. If the execution of a method ends
Execution of programs is modelled as a labelled transition system. A special event, which marks the termination of the whole program, is generated by the \texttt{return} command. The \texttt{noret} command is appended at the end of each thread when the program \texttt{W} starts to run. It generates a special event, which marks the termination of the corresponding thread.

Note that we use one language for both the concrete and abstract specification of externally observable events only. We use \texttt{get\_obs(T)} to represent the subsequence of \texttt{T} consisting of externally observable events only.

\section{Contextual refinement under fair scheduling}

Before we define contextual refinement under fair scheduling, we first introduce some auxiliary definitions in Fig. 5. We use $T_{\omega}$ to represent the subsequence of $T$ that leads to $(W', S')$ with the sequence of externally observable events only. We also insert a \texttt{spawn} event at the head of each event trace to record the number of threads in the program. The lifting $[W]_p$ appends an \texttt{end} command at the end of each thread, which generates a (t, \texttt{term}) event to mark the termination of this thread.

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 $t$ only.

**Definition 1 (Contextual refinement).** $\Pi$ is a contextual refinement of $\Gamma$ under the refinement mapping function $\varphi$ for initial states and under fair scheduling, i.e., $\Pi \sqsubseteq_{\varphi}^{\omega} \Gamma$, iff

\[
\forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, \Sigma_o. \quad (\varphi(\sigma_c) = \Sigma_o) \implies O_{\varphi}[[W, (\sigma_c, \sigma_o, \odot)]] \subseteq O_{\varphi}[[W, (\sigma_c, \Sigma_o, \odot)]] ,
\]

where $\odot = \{ t_1, \ldots, t_n \}$. $W = \text{let } \Pi \text{ in } C_1 \ldots \text{ and } C_n$, and $W = \text{let } \Gamma \text{ in } C_1 \ldots \text{ and } C_n$.

\section{Program Logic}

In this section, we describe our program logic for verifying $\Pi \sqsubseteq_{\varphi}^{\omega} \Gamma$. Below we first show our new assertion language (Sec. IV-A). In particular, we introduce the novel notion of definite actions. The \textbf{TOP} rule (Sec. IV-B) decomposes the goal of verifying $\Pi \sqsubseteq_{\varphi}^{\omega} \Gamma$ into two-phase verification. We verify the definite actions in phase 1 (Sec. IV-C) and verify the termination-preserving refinement in phase 2 (Sec. IV-D).

\subsection{Assertions}

We introduce assertions at different levels in Fig. 6. Relational state assertions (Fig. 6 (b)) specify the relationship between the concrete and abstract states. The full assertion
\[
\begin{align*}
&\text{(let } \Pi \text{ in skip } | \ldots | \text{ skip}, S) \rightarrow (\text{skip}, S) \\
&\text{(let } \Pi \text{ in } C_1 | \ldots | C_n | C_{n+1} \text{ at the concrete level) } \xrightarrow{\text{abort}} \text{ abort}
\end{align*}
\]

(a) program transitions

\[
\begin{align*}
&\text{\(\prod(f) = (y, C) \quad \exists x \in \text{dom}(s_c) \quad \kappa = \{(y \rightarrow n), x, E[\text{skip}]\}\)} \\
&(\text{E}[x := f(E)], ((s_c, h_c), \sigma_o, \sigma_c)) \xrightarrow{\text{(let, } \Pi\text{)}} (C; \text{nore}, ((s_c, h_c), \sigma_o, \sigma_c)) \\
&\kappa = (s_c, x, C) \quad E[S]_{s_c} = n \quad s' = s_c \chi \quad \chi \sim n
\end{align*}
\]

(b) thread transitions

\[
\begin{align*}
&\text{(end, } \chi) \xrightarrow{\text{(term)}} \text{abort} \\
&(C, (s_c \cup s_1, h_o)) \rightarrow (C', (s_c' \cup s_1', h_o')) \quad \text{dom}(s_1) = \text{dom}(s'_1)
\end{align*}
\]

(c) local thread transitions

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

Fig. 4. Selected operational semantics rules.

We also use regular unary state assertions (Fig. 6 (c)) for phase 1 to establish the definition action \(D\) to represent the progress of the concrete object code which do not rely on its environment’s progress. Since \(D\) is a property of the concrete code only, the phase 1 reasoning can be done by traditional unary logic at the concrete level.

### Relational assertions and assertions at the concrete level

The relational state assertions \(P\) and \(Q\) are separation logic assertions over a tuple consisting of the concrete state \(\sigma\), an abstract state \(\Sigma\) and the logical variable mapping \(i \in LVar \rightarrow Int\).

Their semantics is shown at the top of Fig. 7. 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 treat program variables as resources [21] and use \(\text{OWN}(x)\) for the ownership of the program variable \(x\). The assertion \(E_1 \Rightarrow E_2\) specifies a singleton heap at the concrete level with \(E_2\) stored at the address \(E_1\) and requires that the stores contain variables used to evaluate \(E_1\) and \(E_2\). Its counterpart...
(σ, σ', i) \models P \models Q \iff ((σ, i) \models P) \implies ((σ', i) \models Q)
(σ, σ', i) \models (\forall X, D) \iff \forall n. (σ, σ', i(X \sim n)) \models D
(σ, σ', i) \models (D_1 \wedge D_2) \iff ((σ, σ', i) \models D_1) \wedge ((σ, σ', i) \models D_2)
((σ, Σ), (σ', Σ'), i) \models D \iff (σ', σ) \models D

Fig. 8. Semantics of definite actions as rely/guarantee assertions.

for abstract level heaps is represented as \( E_1 \implies E_2 \). \texttt{emp} describes empty stores and heaps at both levels. Semantics of separating conjunction \( P \times Q \) is similar as in separation logic, except that it is now lifted to assertions over relational states \((σ, Σ)\). We also interpret the initial state mapping function \( ϕ \) (used in Definition 1) as a relational assertion. Semantics of the assertion \([p]\) will be defined latter (see Fig. 9).

We describe the interference between the current thread and its environment using rely/guarantee assertions [13]. The relational rely/guarantee assertions \( R \) and \( G \) specify the transitions over the relational states \((σ, Σ)\). Their semantics is defined in the bottom part of Fig. 7. Here we use \( Σ \) for the relational states. A model consists of the initial relational state \( Σ \), the resulting state \( Σ' \) and the logical variable mapping \( i \). We use \([P]\) for identity transitions with the relational states satisfying \( P \). The action \( P \times Q \) says that the initial relational states satisfy \( P \) and the resulting states satisfy \( Q \). Following LRG [4], we introduce separating conjunction over actions to locally define shared state updates. \( G_1 \times G_2 \) means that the actions \( G_1 \) and \( G_2 \) start from disjoint relational states and the resulting states are also disjoint. Here we lift the disjoint union \( \uplus \) over partial mappings to the relational states \( Σ \). The formal definition is omitted. We will explain \([D]\) later. The syntactic sugars \texttt{Id}, \texttt{Emp} and \texttt{True} represent arbitrary identity transitions, empty transitions and arbitrary transitions respectively.

Since we logically split states into local and shared parts as in LRG [4], we need a precise invariant \( I \) to uniquely determine the boundary between local and shared resources. We define the fence \( I \uplus G \) in a similar way as in our previous work [15] and LRG [4], which says that the transition \( G \) must be made within the boundary specified by \( I \) (see Fig. 7). The formal definition of the precise requirement \( \text{Precise}(I) \) is given in TR [16], 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.

The state assertions \( P \) and \( Q \), and the rely/guarantee assertions \( R \) and \( G \) at the concrete level are defined similarly. We use \([P]\) to lift the relational state assertion \( P \) to a concrete state assertion, which ignores the abstract states about \( P \), as defined in Fig. 7. Semantics of other assertions is in TR [16].

**Definite actions.** Fig. 6(c) gives the syntax of the definite actions \( D \). It could be in a “leads-to” form \( P \leadsto Q \) over the state assertions at the concrete level, or with universal quantifications \( \forall X, D \) or in conjunctions \( D_1 \wedge D_2 \). For instance, the definite action for the counter object \texttt{sfInc} in Fig. 1(c) could be defined in the following form to say that \texttt{owner} is eventually incremented.

\[
\forall n. (\ldots \land (\text{owner} = n)) \leadsto (\ldots \land (\text{owner} = n + 1))
\]

The conjunction \( D_1 \wedge D_2 \) is useful when we want to enumerate the definite actions. For instance, when the program uses two locks \( L_1 \) and \( L_2 \) independently, the definite action \( D \) of the whole program is usually in the form of \( D_1 \wedge D_2 \) where \( D_1 \) and \( D_2 \) are the definite actions for \( L_1 \) and \( L_2 \) respectively.

Definite actions are progress properties (i.e., liveness properties). They have much richer semantic meanings than normal actions (i.e., rely/guarantee assertions). For instance, as we explained in Sec. II, having the definite action \( P \leadsto Q \) requires the following in every execution:

1. \( Q \) should eventually hold if \( P \) holds, and
2. \( P \) should be preserved until \( Q \) holds.

It differs from the normal action \( P \times Q \) specifying the state transitions \((σ, σ', i)\), in the following two aspects:

1. \( P \leadsto Q \) describes a duration in an execution, which is usually a trace of state transitions.
2. \( P \leadsto Q \) is satisfied in executions where \( P \) never holds.

We use the action \([D]\) to capture the overall effects of \( D \). As shown in Fig. 8, \( P \leadsto Q \) specifies the transitions where the final states satisfy \( Q \) if the initial states satisfy \( P \). That is, it describes the transitions satisfying \( P \times Q \), which are the “main expected effects” of the definite action, and arbitrary transitions where \( P \) does not hold at the initial states. For \( \forall X, D \) and \( D_1 \wedge D_2 \), we simply overload the meanings of the predicate logic operators \( \forall \) and \( \wedge \).

As an example, consider the following \( D_x \).

\[
\begin{align*}
D_x & \equiv \forall n. ((x = n) \land (n > 0)) \leadsto (x = n + 1) \\
\{D_x\} & \text{describes the state transitions which increment } x \text{ if } x \text{ is positive initially.} \\
\{D_x\} & \text{does not restrict the transitions when initially } x \text{ is not positive.}
\end{align*}
\]

We define some useful syntactic sugars below. \texttt{Enabled}(\( D \)) describes the conditions under which the main expected effects of \( D \) are guaranteed to take place eventually.

\[
\begin{align*}
\text{Enabled}(P \leadsto Q) & \equiv P \\
\text{Enabled}(\forall X, D) & \equiv \exists X. \text{Enabled}(D) \\
\text{Enabled}(D_1 \wedge D_2) & \equiv \text{Enabled}(D_1) \lor \text{Enabled}(D_2)
\end{align*}
\]

Also we use \( \langle D \rangle \) to represent the state transitions of the definite action when it is enabled. That is, \( \langle D \rangle \) specifies the main expected effects of \( D \).

\[
\langle D \rangle \equiv [D] \land (\text{Enabled}(D) \times \text{true})
\]

For the example \( D_x \) defined above, \( \langle D_x \rangle \) can be reduced to the following normal action:

\[
\exists n. ((x = n) \land (n > 0)) \times (x = n + 1)
\]

We define the syntactic sugar \([D]\) as a definite action for the executions before the main expected move \( \langle D \rangle \) takes place.

\[
\begin{align*}
[P \leadsto Q] & \equiv P \leadsto Q \\
[D_1 \wedge D_2] & \equiv [D_1] \wedge [D_2]
\end{align*}
\]

For the example \( D_x \) above, \( [D_x] \) represents the definite action \( \forall n. ((x = n) \land (n > 0)) \leadsto (x = n) \). Then, \( \{[D_x]\} \) specifies the transitions which keep \( x \) unchanged if \( x \) is positive initially.
Full state assertions. In Fig. 6(a) we define the full state assertions $p$ and $q$ for the refinement verification in phase 2. They specify the abstract code and the number of tokens in addition to the relational states.

Semantics of $p$ and $q$ is defined in Fig. 9. Following our earlier work [17], $p$ is interpreted on $(\sigma, w, D, \Sigma, i)$. Here besides the concrete and abstract states $\sigma$ and $\Sigma$ and the logical variable mapping $i$, we introduce some auxiliary data $w$ and $D$. $w$ is the number of tokens needed for loops (see Sec. II). $D$ is either some source code $C$, or a special sign $\bullet$ serving as a unit for defining semantics of $p \ast q$ below.

In Fig. 9 we lift the relational assertion $P$ as a full state assertion to specify the states. $\text{arem}(C)$ says that the remaining abstract code is $C$ at the current program point. $\text{wf}(E)$ states that the number of tokens at the current target code is no less than $E$. We can see $\text{wf}(0)$ always holds, and for any $n$, $\text{wf}(n+1)$ implies $\text{wf}(n)$. We use $[p]_w$ to ignore the descriptions in $p$ about the number of tokens. $[p]_w$ lifts $p$ back to a relational state assertion.

Separating conjunction $p \ast q$ has the standard meaning as in separation logic, which says $p$ and $q$ hold over disjoint parts of $(\sigma, w, D, \Sigma, i)$ respectively (the formal definition elided here). The disjoint union is defined in Fig. 9. Following our earlier work [17], the disjoint union of the numbers of tokens $w_1$ and $w_2$ is the sum of them. The disjoint union of $D_1$ and $D_2$ is defined only if at least one of them is the special sign $\bullet$, which has no knowledge about the remaining abstract code. Therefore we know the following holds (for any $P$ and $C$):

\[
(P \land \text{arem}(C) \land \text{wf}(1)) \ast \text{wf}(1) \land \text{emp} \iff (P \land \text{arem}(C) \land \text{wf}(2))
\]

The stability $\text{Sta}(p, R)$, defined at the bottom of Fig. 9, extends its usual meaning (e.g., see [24], [4]) to the full states.

B. The Top rule

We show the TOP rule in Fig. 10, which allows us to derive the goal $\Gamma \vdash \Pi \models_p \Gamma$ by two-phase verification.

The judgment $\text{Req}, g, J \vdash \{p\} \Pi : D$ for phase 1 establishes the definite action $D$ for the object implementation $\Pi$. Then $\Pi$ is used as one of the inputs of phase 2. The judgment $\Pi, R, G, J \vdash \{P\} \Xi \subseteq \Gamma$ of phase 2 ensures that $\Pi$ refines $\Gamma$ if their initial states satisfy the precondition $P$.

The TOP rule in Fig. 10 requires the proof at the two phases to use consistent specifications. The assertions $\text{Req}, g, J$ and $\mathcal{P}$ for the proof at phase 1 should be the concrete versions lifted from the relational assertions $R, G, I$ and $P$ at phase 2. Moreover, the specification should be well-formed, i.e., $\text{wff}(\varphi, P, D, R, G, I)$ holds. The $\text{wff}$ condition is defined by the SPEC rule, which will be explained later.

In phase 1, as shown by the PHASE1 rule in Fig. 10, we need to verify that every method in the object $\Pi$ establishes the definite action $D$. To reason about the method body, we can use the formal argument $x$ as some extra resource. The pre- and post-conditions are the same here, therefore the assertion $\mathcal{P}$ can be viewed as an object invariant, which must hold at the beginning and the end of each method call.

\[
D := C \mid \bullet
\]

\[
(\sigma, w, D, \Sigma, i) \models P \iff (\sigma, \Sigma, i) \models P
\]

\[
(\sigma, w, D, \Sigma, i) \models \text{arem}(C') \iff D = C'
\]

\[
((s, h), w, D, \Sigma, i) \models \text{wf}(E) \iff \exists n. (\text{wf}(E)(s, h) = n) \land (n \leq w)
\]

\[
(\sigma, w, D, \Sigma, i) \models [p]_w \iff \exists \nu'. (\sigma, w', D, \Sigma, i) \models p
\]

\[
(\sigma, \Sigma, i) \models [p]_w \iff \exists \nu. (\sigma, w, D, \Sigma, i) \models p
\]

\[
(s, h) \equiv (s', h) \iff (s \equiv s', h \equiv h')
\]

\[
D \mid \mathcal{D} \iff (D = \bullet) \lor (\mathcal{D} = \bullet)
\]

\[
D \mid \mathcal{D} \equiv D' \iff \mathcal{D} = \bullet \iff D' = \bullet
\]

\[
(\sigma, w, D, \Sigma, i) \models (\sigma', w', D', \Sigma', i) \equiv (\sigma' \equiv \sigma, w + w', D \equiv D', \Sigma \equiv \Sigma', i), \text{ if } i = i'
\]

\[
\text{Sta}(p, R) \iff \forall \sigma, \sigma', w, D, \Sigma, \Sigma', i. ((\sigma, w, D, \Sigma, i) \models p) \land (\Sigma \equiv \Sigma_p) \implies ((\sigma', w, D, \Sigma', i) \models p)
\]

Similarly, the PHASE2 rule in Fig. 10 requires that each method in $\Pi$ refines its counterpart in $\Gamma$. Starting from initial object states related by $P$ and equivalent arguments $x$ and $y$ at the concrete and the abstract levels, the method body $C$ must fulfill the corresponding abstract code $C$.

The SPEC rule in Fig. 10 defines the $\text{wff}$ requirement about the specifications. First, as in traditional Rely-Guarantee reasoning [13], the guarantee condition of each thread, $G_t$, should imply the rely condition of any other thread, $R_{t'}$, so that the parallel threads $t$ and $t'$ could collaborate. For any thread $t$, the condition $|R_t| \implies \{D_t\}$ requires its environment to preserve its definite action. If $D_t$ is in the form of $\mathcal{P} \Rightarrow Q$, this condition can be reduced to the stability of $\mathcal{P}$ under $|R_t|$. For the example of $\text{sfInc}$ in Fig. 1(e), this condition corresponds to the fact that when the thread $t$ acquires the lock (its definite action is enabled), its environment cannot update owner. We also require $|G_t| \models \{D_t\} \lor |D_t|$. That is, any step of the current thread $t$ should either preserve its definite action or perform the expected move. Besides, the definite action $D_t$ and the precondition $P_t$ should be defined over the shared states precisely determined by the invariant $I$. Also, we require $\text{Enabled}(D_t) \equiv \neg |R_t|$. That is, when $D_t$ is enabled, we must be inside the method body.

C. Inference rules for phase 1

Phase 1 is to verify the definite action $D$. The judgment $\text{Req}, g, J \vdash \{p\} C \{Q, D\}$ says that if $D$ is continuously enabled after some point in an execution of $C$ under the environment interference $\mathcal{R}$, this execution must terminate. Since the specifications satisfy the $\text{wff}$ requirement discussed earlier, we know the execution either guarantees the definite action (i.e., $(D)$ eventually takes place whenever $D$ becomes enabled), or terminates at a state where $D$ is enabled. The latter case cannot occur because $D$ is enabled only inside the method body (recall the premise $\text{Enabled}(D_t) \Rightarrow \neg |R_t|$ in the SPEC
\[
\frac{\{R\}, \{G\}, \{I\} \vdash \{P\} \Pi : D}{\mathcal{D}, R, G, I \vdash \{P\} \Pi \subseteq \Gamma} \quad \text{(TOP)}
\]

for all \( f \in \text{dom}(\Pi) : \quad \Pi(f) = (x, C) \quad \mathcal{R}_f, G, J \vdash \{p \ast \text{own}(x)\} C ; \text{notet}(\{p \ast \text{own}(x), D\})
\]

\[
\frac{\mathcal{R}_f, G, J \vdash \{p\} \Pi : D}{D, R, G, I \vdash \{P\} \Pi \subseteq \Gamma} \quad \text{(PHASE1)}
\]

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

\[
\frac{D, R, G, I \vdash \{P \ast \text{own}(x) \ast \text{own}(y) \land (x = y) \land \text{arem}(C)\} C ; \text{notet}(\{P \ast \text{own}(x) \ast \text{own}(y) \land \text{arem}(\text{skip})\})}{D, R, G, I \vdash \{P\} \Pi \subseteq \Gamma} \quad \text{(PHASE2)}
\]

for all \( t, t' \) such that \( t \neq t' : \quad G_t \Rightarrow R_{t'}
\]

\[
\frac{\mathcal{R}_f, G, J \vdash \{p\} \Pi : D}{D, R, G, I \vdash \{P\} \Pi \subseteq \Gamma} \quad \text{(SPEC)}
\]

\[
\frac{[3], G, J \vdash \{p\}(C)(\{Q, D\})}{\mathcal{R}_f, G, J \vdash \{p\} \text{while } (B)(C)(\{Q \land \neg B, D\})} \quad \text{(ATOM-R)}
\]

\[
\frac{\text{Sta}(\{p, Q\}, \mathcal{R}_f \ast \text{id}) \triangleright \triangleright \mathcal{R}_f}{[3], G, J \vdash \{p\} \text{while } (B)(C)(\{Q, D\})} \quad \text{(ATOM)}
\]

\[
\frac{\mathcal{D}, R, G, I \vdash \{p \land B\} C(p)}{\mathcal{D}, R, G, I \vdash \{p\} \text{while } (B)(C){\{p \land \neg B\}}} \quad \text{(WHILE)}
\]

\[
\frac{\mathcal{D}, R, G, I \vdash \{p\} C(q)}{\mathcal{D}, R, G, I \vdash \{p\} C(q)} \quad \text{(HIDE-w)}
\]

Fig. 10. Top rule.

Fig. 11. Inference rules for phase 1.

\[
\frac{\mathcal{D}, R, G, I \vdash \{p \land B\} C(p)}{\mathcal{D}, R, G, I \vdash \{p\} \text{while } (B)(C){\{p \land \neg B\}}} \quad \text{(WHILE)}
\]

\[
\frac{\mathcal{D}, R, G, I \vdash \{p\} C(q)}{\mathcal{D}, R, G, I \vdash \{p\} C(q)} \quad \text{(HIDE-w)}
\]

Fig. 12. Inference rules for phase 2.

\[
\frac{[3], G, J \vdash \{p\} \Pi : D}{\mathcal{D}, R, G, I \vdash \{p\} \Pi \subseteq \Gamma} \quad \text{(SPEC)}
\]

\[
\frac{\mathcal{D}, R, G, I \vdash \{p\} \Pi : D}{\mathcal{D}, R, G, I \vdash \{p\} \Pi \subseteq \Gamma} \quad \text{(SPEC)}
\]

\[
\frac{\mathcal{D}, R, G, I \vdash \{p\} C(q)}{\mathcal{D}, R, G, I \vdash \{p\} C(q)} \quad \text{(HIDE-w)}
\]

Fig. 12. Inference rules for phase 2.

rule). Then we can conclude that the method implementation guarantees the definite action.

Fig. 11 shows the key inference rules for phase 1, which allow us to derive \( \mathcal{R}_f, G, J \vdash \{p\} C(C, D) \). Other rules are straightforward extensions to LRG [4] and given in TR [16].

The WHILE rule proves the termination of the loop if \( D \) is continuously enabled. We have informally explained its idea in Sec. II. In detail, we find a metric \( E \) which can be evaluated at the local parts of the states satisfying \( \mathcal{P} \) (required by the third premise \( \mathcal{P} \Rightarrow (E = E) \ast J \)). The second premise shows that for each iteration, if initially \( \text{Enabled}(D) \) holds at the shared state, then \( E \) should decrease at the end of the iteration. Here \( X \) is not free in the code \( C \) and the specifications, i.e., \( X \) is a fresh logical variable, as required by the last premise. From the fourth premise, we know the loop terminates if \( \text{Enabled}(D) \) continuously holds. Besides, the first premise gives us the correctness of the loop body \( C \) for all loop iterations (no matter \( D \) is enabled or not).

The ATOM rule allows us to reason sequentially about the code in the atomic block. We use \( \vdash \{p\} C[q] \) to represent the total correctness of \( C \) in sequential separation logic. The corresponding rules are standard and elided here. We can lift \( C \)’s total correctness to the concurrent setting as long as its overall transition over the shared states satisfies the guarantee \( \mathcal{G} \). Here we assume the environment is identity transitions. To allow general environment behaviors, we can apply the ATOM-R rule later, which requires that \( \mathcal{R} \) be fenced by \( J \) and the pre- and post-conditions be stable with respect to \( \mathcal{R} \).

D. Inference rules for phase 2

Phase 2 is to verify the refinement relation between the concrete method code \( C \) and its abstract operation \( C \). The judgment \( D, R, G, I \vdash \{p \land \text{arem(C)}\} C\{Q \land \text{arem(skip)}\} \) establishes a simulation relation between \( C \) and \( C \), which ensures the preservation of termination when the environment interference guarantees the definite action \( D \).

Fig. 12 shows some key inference rules for phase 2, which allow us to derive \( \mathcal{D}, R, G, I \vdash \{p\} C[q] \).

As we explained in Sec. II, the WHILE rule needs us to find two special assertions \( J \) and \( Q \) over the shared states. When \( Q \) holds, the loop iterations should guarantee to preserve the termination of the abstract code. We decrease the number of tokens at the beginning of each such loop iteration and re-establish \( p \) between the states and the number of tokens at the end of such iteration. That is, either we re-gain the tokens by fulfilling some abstract operations during the iteration, or the loop gets closer to its termination. The premises \( p \Rightarrow J \ast \text{true} \) and \( \text{Sta}(J, R \lor G) \) ensure that \( J \) is an invariant preserved at every step of the execution. The condition \( R, G : J \oplus f \Rightarrow \top \) ensures that \( Q \) will be eventually reached from any state satisfying \( J \) under the definite action \( D \) made by the environment. It is formally defined as follows.
Definition 2 (Definite progress). \( R, G : J @ f \stackrel{D}{\rightarrow} Q \) iff for any \( t, \mathcal{S} \) and \( i \), if \( (\mathcal{S}, i) \models J_t \), then both the following hold:

1. either \( (\mathcal{S}, i) \models Q_t \) holds,
2. or both the following hold:
   - \( (\mathcal{S}, i) \models \exists t' \neq t. \text{Enabled}(D_t) \); and
   - for any \( t' \neq t \) and \( \mathcal{S}' \), if \( (\mathcal{S}', \mathcal{S}', i) \models (D_{t'}) \land R_t \), then \( f(\mathcal{S}', i) < f(\mathcal{S}, i) \).

Here \( f \) is a function that maps the states \( (\mathcal{S}, i) \) to some metrics over which there is a well-founded order \(<\).

That is, when \( J \) holds, either \( Q \) holds, or the definite action of some environment thread \( t' \) is enabled. For the latter case, we require the metric \( f(\mathcal{S}, i) \) to decrease after the transitions \( (D_t) \). Besides, the metric should never increase at any step of the execution.

The ATOM rule allows us to logically execute the abstract code simultaneously with the concrete atomic step. We use \( p \Rightarrow^* q \) for the multi-step executions from the abstract code specified by \( p \) to the code specified by \( q \), which is defined in the bottom part of Fig. 9. We also write \( p \Rightarrow^+ q \) to say that either \( p \Rightarrow^* q \) or the usual implication \( p \Rightarrow q \) holds. Then, the ATOM rule says, we can execute zero-or-more steps of the abstract code before or after the steps of \( C \), as long as the overall transition (including the abstract steps and the concrete steps) satisfies the relational guarantee \( G \). The ATOM-R rule for phase 2 is similar to the one for phase 1 and omitted here.

We also provide the HIDE-W rule to discard the tokens (by using \( \llbracket m \rrbracket \)) when the termination-preservation of code \( C \) is already established.

E. Soundness

Our logic is a sound proof technique for the contextual refinement under fair scheduling (Definition 1), as shown by the following theorem.

Theorem 3 (Soundness). If \( \vdash \Pi \sqsubseteq^G \Gamma \), then \( \Pi \sqsubseteq^{G'} \Gamma \).

The detailed soundness proofs are given in TR [16].

V. Linearizability, Starvation-freedom and Contextual Refinement

Below we show the contextual refinement \( \Pi \sqsubseteq^{G'} \Gamma \) is equivalent to linearizability and starvation-freedom together. Then our logic can be viewed as a proof method for verifying linearizable objects.

Linearizability describes atomic behaviors of object implementations. Following its standard definition [11], we define linearizability using histories, which are finite event traces \( T \) consisting of only method invocations, returns, and object faults. We say a return \( e_2 \) matches an invocation \( e_1 \), denoted as \( \text{match}(e_1, e_2) \), iff they have the same thread ID. An invocation is pending in \( T \) if no matching return follows it. We use \( \text{pend}_{\text{inv}}(T) \) 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 completions(\( T \)). It is a set of histories, and for each history in it, every invocation has a matching return event.

Definition 4 (Linearizable histories). \( T \sqsubseteq_{\text{lin}} T' \) iff

1. \( \forall t. T_t = T'_t \); and
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 object is linearizable iff each of its concurrent histories after completions is linearizable with respect to some legal sequential history. We use \( \Gamma \sqsupseteq (\Sigma_o, T') \) to mean that \( T' \) is a legal sequential history generated by any client using the specification \( \Gamma \) with an abstract initial state \( \Sigma_o \).

Definition 5 (Linearizable objects). \( \Pi \) is linearizable with respect to \( \Gamma \) under the initial state mapping \( \varphi \). \( \Pi \sqsubseteq^{\varphi} \Gamma \), iff

\[ \forall n, C_n, \sigma_c, \sigma_o, \Sigma_o. T \in H(\{\text{let } \Pi \text{ in } C_1 \ldots \ldots C_n\}, (\sigma_c, \sigma_o, \emptyset)) \land (\varphi(\Sigma_o) = \Sigma_o) \implies \exists T'. T_o \in \text{completions}(T) \land \Gamma \sqsupseteq (\Sigma_o, T') \land T_e \sqsubseteq_{\text{lin}} T' \]

Starvation-freedom [10] requires that in fair executions every method call eventually return. We define \( \text{prog-t}(T) \) to say that every method call in \( T \) eventually finishes.

\[ \text{prog-t}(T) \text{ iff } \forall i, e. e \in \text{pend}_{\text{inv}}(T_{1..i}) \implies \exists j > i \land \text{match}(e, T(j)) \]

We write \( \text{abt}(T) \) to say that \( T \) ends with a fault event.

Definition 6 (Starvation-free objects). \( \text{starvation-free}_{\varphi}(\Pi) \text{ iff } \forall n, C_n, \sigma_c, \sigma_o, T. (T \in T_o[W; (\sigma_c, \sigma_o, \emptyset)] \land (\sigma_o \in \text{dom}(\varphi)) \land \text{fair}(T) \implies \text{prog-t}(T) \lor \text{abt}(T) , \) \]

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

Suppose each method body in \( \Gamma \) is an atomic operation of the form \( \langle C \rangle \); \text{return } E, and such \( \langle C \rangle \) always terminates. We have the following equivalence result. The proof is in TR [16].

Theorem 7 (Equivalence).

\( \Pi \sqsubseteq^{\varphi} \Gamma \land \text{starvation-free}_{\varphi}(\Pi) \iff \Pi \sqsubseteq^{G'} \Gamma \).

VI. Examples

We successfully apply our logic to verify simple objects with coarse-grained synchronization using ticket lock [19] and various queue locks (including Anderson array-based lock, CLH lock and MCS lock) [9]. As examples with more permissive locking scheme, we verify the Michael-Scott two-lock queue algorithm [20] and the lock-coupling list algorithm [9]. Although the examples verified here are all starvation-free, our logic is not limited only to verifying linearizability and starvation-freedom. It supports verification of general refinement under fair scheduling, where the abstract specification can be non-atomic. Moreover, we expect deadlock-free objects and lock-free objects would also satisfy the refinement under fair scheduling given proper (non-atomic) specifications.

Below we explain the key proofs for the counter object with ticket lock (\text{sfInc} in Fig. 1(c)), and discuss the verification
Fig. 13. The counter using ticket lock, sfInc, with auxiliary code for proofs.

\( I \models \exists n_1, n_2. \ \text{lock}(n_1, n_2) \land (\text{wait}_{n_1} \land \text{resource} \lor \neg \text{wait}_{n_1}) \land \text{waits}(n_1, n_2) \)

\( G_1 \models (\text{ReqLock} \lor \text{AcqResource} \lor \text{RelLock})(\text{id} \land (I \land 1)) \)

\( \text{ReqLock} \models \exists \text{id}, n_1, n_2. (\text{lock}(\text{id}, n_1, n_2) \land (\neg \text{wait}_{n_2})) \land (\text{locked}(\text{id}, n_1, n_2 + 1) \land \text{wait}_{n_2}) \)

\( \text{AcqResource} \models \exists \text{id}, n_1, n_2. (\text{locked}(\text{id}, n_1, n_2) \land \text{wait}_{n_1} \land \text{resource}) \land (\text{locked}(\text{id}, n_1, n_2) \land (\neg \text{wait}_{n_1})) \)

\( \text{RelLock} \models \exists \text{id}, n_1, n_2, (\text{locked}(\text{id}, n_1, n_2)) \land (\text{locked}(\text{id}, n_1, n_2) \land (\text{wait}_{n_1} \land \text{resource}) \land (\text{locked}(\text{id}, n_1, n_2) \land (\neg \text{wait}_{n_1}))) \)

\( D_0 \models \forall n_1. \ dp(n_1) \Rightarrow dp(n_1) \)

\( dp(n_1) \models \exists \text{id}, n_2 \land \text{locked}(\text{id}, n_1, n_2) \land \text{true} \land I \)

\( \text{RelLock} \models \exists \text{id}, n_1, n_2, \text{locked}(\text{id}, n_1, n_2) \land (\neg \text{wait}_{n_2}) \land (\text{resource} \lor (\neg \text{wait}_{n_2}) \land (n_1 < X)) \land \text{waits}(n_1, n_2) \)

\( Q_h \models \exists n_2, X, \text{locked}(\text{id}, n_2, X) \land \text{wait}_{n_2} \land \text{resource} \land \text{waits}(X, n_2) \)

Fig. 14. Auxiliary definitions for the counter with ticket lock.

of the lock-coupling list algorithm. The full proofs for all the examples we verified are in TR [16].

The counter with ticket lock. As we explained in Sec. II, sfInc in Fig. 1(c) is linearizable with respect to INC because sfInc 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. sfInc 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. 13, we introduce the explicit tickets ticket, ticket_1, ..., Each ticket_i 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_i to indicate whether the thread ticket_i is requesting the lock or has entered its critical section. It helps describe the ownership transfer over the resource x. Initially, wait_i is false. It is set to true when the thread gets its ticket at line 2. After line 3, wait_i is reset to transfer the shared resource x to the thread so that the thread can freely access x in the critical section (line 5).

Fig. 14 defines the precise invariant I which determines the boundaries of shared states. A shared state contains the lock (with owner = n_1 and next = n_2), the wait bits (waits) and the protected resource if wait_owner holds. Here resource requires x to be the same at the concrete and the abstract sides. lock(tl, n_1, n_2) contains the auxiliary tickets in addition to owner and next, where tl (hidden in the definition of I) is the list of the threads ticket_owner, ..., ticket_next-1. We also use locked(tl, n_1, n_2) for the case when tl is not empty. That is, the lock is acquired by the first thread in tl, while the other threads in tl are waiting for the lock in order. Besides, we use lockIr_run(tl, n_1, n_2) short for lock(tl, n_1, n_2) ∧ (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 requires the lock to satisfy \( \exists dl. \text{lockIr}_{dl}(t, n_1, n_2) \).

The guarantee condition G in Fig. 14 defines the atomic actions of a thread t. ReqLock_i adds the thread t at the end of t of the threads requesting the lock and increments next. It corresponds to line 2 of Fig. 13. AcqResource transfers the resource to the thread t when it has acquired the lock, corresponding to line 4 of Fig. 13. RelLock_i 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 6 of Fig. 13. The rely condition R include all the atomic actions made by the environment threads.

Next we define the definite action D in Fig. 14. As we explained in Sec. II, the thread t’s definite action \( \text{D}_t \) requires that whenever t holds a lock with owner = n_1 (specified by \( dp(n_1) \)), it should eventually release the lock by incrementing owner to n_1 + 1 (specified by \( dp(n_1) \)). Note that \( dp(n_1) \) 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 4 but its loop at line 3 must terminate. Besides, \( dp(n_1) \) is stable under the environment which may enqueue more threads into the list tl of waiting threads. We can prove \( [R_0] \Rightarrow \{[D_1]\} \) and \( [G_1] \Rightarrow \{[D_1]\} \lor [D_2]\), which are required by the wff condition at the top rule in Fig. 10.

With the above specifications, we reason about the code of Fig. 13 in two phases. The proof of phase 1 is much like the usual proof for partial correctness, except that we also prove the termination of the loop at line 3 when D is enabled.

For phase 2, to verify the loop at line 3, we first define J and Q in Fig. 14. Both J and Q are stronger than I. For thread t, J says t is requesting the lock. Here locked(tl_1,tl_2)(n_1, n_2) says (1) t takes the ticket number X, which satisfies n_1 = owner ≤ X < next = n_2, and (2) the threads requesting the lock before and after t constitute the lists tl_1 and tl_2 respectively. Then, the first thread of the whole concatenated list tl_1 ∪ tl_2 holds the lock. Q specifies the case when tl_1 is empty (thus X = owner). The loop terminates when Q holds. We also strengthen the guarantee of the loop to \( G' \Leftrightarrow [I], \) the identity transitions. We have \( \text{Sta}(\{J, Q\}, G' \lor R) \).

Next we define f and prove R, G_’ : J@f \( \Rightarrow Q \). The metric function f maps each shared state (S, i) 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 = i holds). Since J ensures owner ≤ X, we can use the usual order on natural numbers as the associated well-founded order. The condition R, G_’ : J@f \( \Rightarrow Q \) 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 \((X \rightarrow \text{owner})\) 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 sfInc with respect to the atomic INC under fair scheduling. By Theorem 7, we get linearizability and starvation-freedom of sfInc.

The lock-coupling list. The algorithm implements an abstract set with \( \text{add} \) and \( \text{rmv} \) operations [9]. The concrete list is an ordered singly-linked list with the \textbf{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.

We verify a version of the algorithm where the locks of the list nodes are implemented using ticket lock. The difficulty lies in defining the definite action. Unlike the counter object, in lock-coupling list, the thread lies in defining the definite action. Unlike the counter object, list nodes are implemented using ticket lock. The difficulty both the node to be removed and its predecessor are locked. The lock of the successor node. In fact, the progress of the thread may first try to acquire the lock because the successor is locked.

add(e)
hand locking: the lock on one node is not released until its successor is locked. The smallest and the biggest values respectively. Each list node
sentinel nodes at the two ends of the list containing the Head
The lock-coupling list.
The algorithm implements an abstract
add
"π" with re-
back and Xu [2] and Henzinger et al. [8] propose sim-
progress of other threads. Here we address the problem of helping which commonly appears in starvation-free objects: the progress of a thread relies on the helps from other threads. Besides, they support multiple layers while our two-phase logic allows induction proofs. (Actually Hoffmann et al. [12] show that the delaying problem can be solved by verification in a single pass, thus the layered proofs are not necessary.)

Our notion of definite action \( D \) is inspired by the “leads-to” formulas in TLA [14]. But we do not use temporal formulas to reason about execution traces. In our logic, we mainly use \( \{D\} \) and \( \{\{D\}\} \) which define the behaviors of \( D \) over state transitions instead of the whole execution traces.

In our previous work [18], we prove that linearizability and starvation-freedom together is equivalent to a contextual refinement which assumes fair scheduling at the concrete level only. Here we define a new contextual refinement \( \Pi \sqsubseteq_{\text{l}}^{\text{f}} \Gamma \) which is also equivalent to linearizability and starvation-freedom, but it now assumes fairness at both the concrete and abstract levels. The new refinement definition enables transitivity, thus is probably more useful in practice.

VII. RELATED WORK

There is a large body of work on verifying fair termination (i.e., termination or total correctness of programs under fair scheduling), which may date back to Apt and Olderog [1], Grumberg et al. [7] and Stølen [22]. In this paper, we verify a contextual refinement under fair scheduling, which is more like a property about the preservation of fair termination. If we know \( \Pi \sqsubseteq_{\text{l}}^{\text{f}} \Gamma \) (verified in our logic), then any client program that fairly terminates with \( \Gamma \) (verified in earlier work) is guaranteed to also fairly terminate using \( \Pi \) instead.

Back and Xu [2] and Henzinger et al. [8] propose simulations to verify refinement under fair scheduling. Their simulations are not thread-local, while a key feature of our logic (and the main challenge we address) is about the support of thread-local verification.

Gotsman et al. [6] verify lock-freedom using layered proofs. Their work addresses the problem of delaying in lock-free objects: the progress of a thread may be delayed by the progress of other threads. Here we address the problem of helping which commonly appears in starvation-free objects: the progress of a thread relies on the helps from other threads. Besides, they support multiple layers while our two-phase logic allows induction proofs. (Actually Hoffmann et al. [12] show that the delaying problem can be solved by verification in a single pass, thus the layered proofs are not necessary.)

REFERENCES