Modular Verification of Linearizability with Non-Fixed Linearization Points

Hongjin Liang Xinyu Feng

University of Science and Technology of China

lhj1018@mail.ustc.edu.cn xyfeng@ustc.edu.cn

Abstract

Locating linearization points (LPs) is an intuitive approach for proving linearizability, but it is difficult to apply the idea in Hoare-style logic for formal program verification, especially for verifying algorithms whose LPs cannot be statically located in the code. In this paper, we propose a program logic with a lightweight instrumentation mechanism which can verify algorithms with non-fixed LPs, including the most challenging ones that use the helping mechanism to achieve lock-freedom (as in HSY elimination-based stack), or have LPs depending on unpredictable future executions (as in the lazy set algorithm), or involve both features. We also develop a thread-local simulation as the meta-theory of our logic, and show it implies contextual refinement, which is equivalent to linearizability. Using our logic we have successfully verified various classic algorithms, some of which are used in the `java.util.concurrent` package.

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

General Terms Theory, Verification

Keywords Concurrency; Rely-Guarantee Reasoning; Linearizability; Refinement; Simulation

1. Introduction

Linearizability is a standard correctness criterion for concurrent object implementations [16]. It requires the fine-grained implementation of an object operation to have the same effect with an instantaneous atomic operation. To prove linearizability, the most intuitive approach is to find a linearization point (LP) in the code of the implementation, and show that it is the single point where the effect of the operation takes place.

However, it is difficult to apply this idea when the LPs are not fixed in the code of object methods. For a large class of lock-free algorithms with *helping* mechanism (*e.g.*, HSY elimination-based stack [14]), the LP of one method might be in the code of some other method. In these algorithms, each thread maintains a descriptor recording all the information required to fulfill its intended operation. When a thread A detects conflicts with another thread B, A may access B's descriptor and help B finish its intended operation first before finishing its own. In this case, B's operation takes effect at a step from A. Thus its LP should *not* be in its own code, but in the code of thread A.

Besides, in optimistic algorithms and lazy algorithms (*e.g.*, Heller *et al.*'s lazy set [13]), the LPs might depend on unpredictable future interleavings. In those algorithms, a thread may access the shared states as if no interference would occur, and validate the accesses later. If the validation succeeds, it finishes the operation; otherwise it rolls back and retries. Its LP is usually at a prior state access, but only if the later validation succeeds.

Reasoning about algorithms with non-fixed LPs has been a long-standing problem. Most existing work either supports only simple objects with static LPs in the implementation code (*e.g.*, [2, 5, 19, 30]), or lacks formal soundness arguments (*e.g.*, [32]). In this paper, we propose a program logic for verification of linearizability with non-fixed LPs. For a concrete implementation of an object method, we treat the corresponding abstract atomic operation and the abstract state as auxiliary states, and insert auxiliary commands at the LP to execute the abstract operation simultaneously with the concrete step. We verify the instrumented implementation in an existing concurrent program logic (we will use LRG [8] in this paper), but extend it with new logic rules for the auxiliary commands. We also give a new relational interpretation to the logic assertions, and show that at the LP, the step of the original concrete implementation has the same effect as the abstract operation. We handle non-fixed LPs in the following way:

- To support the helping mechanism, we collect a pending thread pool as auxiliary state, which is a set of threads and their abstract operations that might be helped. We allow the thread that is currently being verified to use auxiliary commands to help execute the abstract operations in the pending thread pool.
- For future-dependent LPs, we introduce a try-commit mechanism to reason with uncertainty. The **try** clause guesses whether the corresponding abstract operation should be executed and keeps all possibilities, while **commit** chooses a specific possible case when we know which guess is correct later.

Although our program logic looks intuitive, it is challenging to prove that the logic is sound *w.r.t.* linearizability. Recent work has shown the equivalence between linearizability and contextual refinement [5, 9, 10]. The latter is often verified by proving simulations between the concrete implementation and the atomic operation [5]. The simulation establishes some correspondence between the executions of the two sides, showing there exists one step in the concrete execution that fulfills the abstract operation. Given the equivalence between linearizability and refinement, we would expect the simulations to justify the soundness of the LP method and to serve as the meta-theory of our logic. However, existing thread-local simulations do not support non-fixed LPs (except the recent work [31], which we will discuss in Sec. 7). We will explain the challenges in detail in Sec. 2.

Our work is inspired by the earlier work on linearizability verification, in particular the use of auxiliary code and states by Vafeiadis [32] and our previous work on thread-local simulation RGSim [19], but makes the following new contributions:

- We propose the first program logic that has a formal soundness proof for linearizability with non-fixed LPs. Our logic is built upon the unary program logic LRG [8], but we give a relational interpretation of assertions and rely/guarantee conditions. We also introduce new logic rules for auxiliary commands used specifically for linearizability proofs.
- We give a light instrumentation mechanism to relate concrete implementations with abstract operations. The systematic use of auxiliary states and commands makes it possible to execute the abstract operations synchronously with the concrete code. The try-commit clauses allow us to reason about future-dependent uncertainty without resorting to prophecy variables [1, 32], whose existing semantics (*e.g.*, [1]) is unsuitable for Hoare-style verification.
- We design a novel thread-local simulation as the meta-theory for our logic. It generalizes RGSim [19] and other compositional reasoning of refinement (*e.g.*, [5, 30]) with the support for non-fixed LPs.
- Instead of ensuring linearizability directly, the program logic and the simulation both establish contextual refinement, which we prove is equivalent to linearizability. A program logic for contextual refinement is interesting in its own right, since contextual refinement is also a widely accepted (and probably more natural) correctness criterion for library code.
- We successfully apply our logic to verify 12 well-known algorithms. Some of them are used in the `java.util.concurrent` package, such as MS non-blocking queue [23] and Harris-Michael lock-free list [11, 22].

In the rest of this paper, we first analyze the challenges in the logic design and explain our approach informally in Sec. 2. Then we give the basic technical setting in Sec. 3, including a formal operational definition of linearizability. We present our program logic in Sec. 4, and the new simulation relation as the meta-theory in Sec. 5. In Sec. 6 we summarize all the algorithms we have verified and sketch the proofs of three representative algorithms. We discuss related work and conclude in Sec. 7.

2. Challenges and Our Approach

Below we start from a simple program logic for linearizability with fixed LPs, and extend it to support algorithms with non-fixed LPs. We also discuss the problems with the underlying meta-theory, which establishes the soundness of the logic *w.r.t.* linearizability.

2.1 Basic Logic for Fixed LPs

We first show a simple and intuitive logic which follows the LP approach. As a working example, Fig. 1(a) shows the implementation of push in Treiber stack [29] (let's first ignore the blue code at line `7'`). The stack object is implemented as a linked list pointed to by `S`, and `push(v)` repeatedly tries to update `S` to point to the new node using compare-and-swap (`cas`) until it succeeds.

To verify linearizability, we first locate the LP in the code. The LP of `push(v)` is at the `cas` statement when it succeeds (line `7`). That is, the successful `cas` can correspond to the abstract atomic `PUSH(v)` operation: `Stk := v::Stk`; and all the other concrete steps cannot. Here we simply represent the abstract stack `Stk` as

```
1  push(int v) {
2   local x, t, b;
3   x := new node(v);
4   do {
5     t := S;
6     x.next := t;
7     <b := cas(&S,t,x);
7'     if(b) linself;>
8   } while(!b);
9  }
```

(a) Treiber Stack

```
1  readPair(int i, j) {
2   local a, b, v, w;
3   while(true) {
4     <a := m[i].d; v := m[i].v;>
5     <b := m[j].d; w := m[j].v;
5'     trylinself;>
6     if(v = m[i].v) {
6'       commit(cid ↣ (end, (a, b)));
7       return (a, b); }
8   } }
9  write(int i, d) {
10   <m[i].d := d; m[i].v++;> }
```

(c) Pair Snapshot

```
1   push(int v) {
2    local p, him, q;
3    p := new thrdDescriptor(cid, PUSH, v);
4    while(true) {
5      if (tryPush(v)) return;
6      loc[cid] := p;
7      him := rand(); q := loc[him];
8      if (q != null && q.id = him && q.op = POP)
9        if (cas(&loc[cid], p, null)) {
10         <b := cas(&loc[him], q, p);
10'          if(b) {lin(cid); lin(him);}>
11         if (b) return; }
12     ...
13   } }
```

(b) HSY Elimination-Based Stack

Figure 1. LPs and Instrumented Auxiliary Commands

a sequence of values with "`::`" for concatenation. Then `push(v)` can be linearized at the successful `cas` since it is the single point where the operation takes effect.

We can encode the above reasoning in an existing (unary) concurrent program logic, such as Rely-Guarantee reasoning [17] and CSL [24]. Inspired by Vafeiadis [32], we embed the abstract operation γ and the abstract state θ as auxiliary states on the concrete side, so the program state now becomes $(\sigma, (\gamma, \theta))$, where σ is the original concrete state. Then we instrument the concrete implementation with an auxiliary command **linself** (shorthand for "linearize self") at the LP to update the auxiliary state. Intuitively, **linself** will execute the abstract operation γ over the abstract state θ, as described in the following operational semantics rule:

$$\frac{(\gamma, \theta) \rightsquigarrow (\mathbf{end}, \theta')}{(\mathbf{linself}, (\sigma, (\gamma, \theta))) \longrightarrow (\mathbf{skip}, (\sigma, (\mathbf{end}, \theta')))}$$

Here $\rightsquigarrow$ encodes the transition of γ at the abstract level, and **end** is a termination marker. We insert **linself** into the same atomic block with the concrete statement at the LP, such as line `7'` in Fig. 1(a), so that the concrete and abstract sides are executed simultaneously. Here the atomic block $\langle C \rangle$ means C is executed atomically. Then we reason about the instrumented code using a traditional concurrent logic extended with a new inference rule for **linself**.

The idea is intuitive, but it cannot handle more advanced algorithms with non-fixed LPs, including the algorithms with the helping mechanism and those whose locations of LPs depend on the future interleavings. Below we analyze the two challenges in detail and explain our solutions using two representative algorithms, the HSY stack and the pair snapshot.

2.2 Support Helping Mechanism with Pending Thread Pool

HSY elimination-based stack [14] is a typical example using the helping mechanism. Figure 1(b) shows part of its push method implementation. The basic idea behind the algorithm is to let a push and a pop cancel out each other.

At the beginning of the method in Fig. 1(b), the thread allocates its *thread descriptor* (line 3), which contains the thread id, the name of the operation to be performed, and the argument. The current thread cid first tries to perform Treiber stack's push (line 5). It returns if succeeds. Otherwise, it writes its descriptor in the global loc array (line 6) to allow other threads to eliminate its push. The elimination array loc$[1..n]$ has one slot for each thread, which holds the pointer to a thread descriptor. The thread randomly reads a slot him in loc (line 7). If the descriptor q says him is doing pop, cid tries to eliminate itself with him by two cas instructions. The first clears cid's entry in loc so that no other thread could eliminate with cid (line 9). The second attempts to mark the entry of him in loc as "eliminated with cid" (line 10). If successful, it should be the LPs of *both* the push of cid and the pop of him, with the push happening immediately before the pop.

The helping mechanism allows the current thread to linearize the operations of other threads, which cannot be expressed in the basic logic. It also breaks modularity and makes thread-local verification difficult. For the thread cid, its concrete step could correspond to the steps of both cid and him at the abstract level. For him, a step from its environment could fulfill its abstract operation. We must ensure in the thread-local verification that the two threads cid and him always take consistent views on whether and how the abstract operation of him is done. For example, if we let a concrete step in cid fulfill the abstract pop of him, we must know him is indeed doing pop and its pop has not been done before. Otherwise, we will not be able to compose cid and him in parallel.

We extend the basic logic to express the helping mechanism. First we introduce a new auxiliary command **lin**(t) to linearize a specific thread t. For instance, in Fig. 1(b) we insert line 10' at the LP to execute both the push of cid and the pop of him at the abstract level. We also extend the auxiliary state to record both abstract operations of cid and him. More generally, we embed a pending thread pool U, which maps threads to their abstract operations. It specifies a set of threads whose operations might be helped by others. Then under the new state $(\sigma, (U, \theta))$, the semantics of **lin**(t) just executes the thread t's abstract operation in U, similarly to the semantics of **linself** discussed before.

The shared pending thread pool U allows us to recover the thread modularity when verifying the helping mechanism. A concrete step of cid could fulfill the operation of him in U as well as its own abstract operation; and conversely, the thread him running in parallel could check U to know if its operation has been finished by others (such as cid) or not. We gain consistent abstract information of other threads in the thread-local verification. Note that the need of U itself does not break modularity because the required information of other threads' abstract operations can be inferred from the concrete state. In the HSY stack example, we know him is doing pop by looking at its thread descriptor in the elimination array. In this case U can be viewed as an abstract representation of the elimination array.

2.3 Try-Commit Commands for Future-Dependent LPs

Another challenge is to reason about optimistic algorithms whose LPs depend on the future interleavings.

We give a toy example, pair snapshot [27], in Fig. 1(c). The object is an array m, each slot of which contains two fields: d for the data and v for the version number. The write(i,d) method (lines 9) updates the data stored at address i and increments the version number instantaneously. The readPair(i,j) method intends to perform an atomic read of two slots i and j in the presence of concurrent writes. It reads the data at slots i and j separately at lines 4 and 5, and validate the first read at line 6. If i's version number has not been increased, the thread knows that when it read j's data at line 5, i's data had not been updated. This means the two reads were at a consistent state, thus the thread can return. We can see that the LP of readPair should be at line 5 when the thread

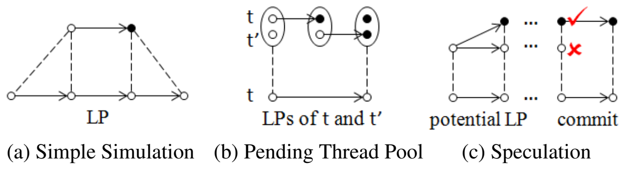

Figure 2. Simulation Diagrams

reads j's data, but only if the validation at line 6 succeeds. That is, whether we should linearize the operation at line 5 depends on the future unpredictable behavior of line 6.

As discussed a lot in previous work (*e.g.*, [1, 32]), the future-dependent LPs cannot be handled by introducing history variables, which are auxiliary variables storing values or events in the past executions. We have to refer to events coming from the unpredictable future. Thus people propose prophecy variables [1, 32] as the dual of history variables to store future behaviors. But as far as we know, there is no semantics of prophecy variables suitable for Hoare-style local and compositional reasoning.

Instead of resorting to prophecy variables, we follow the speculation idea [31]. For the concrete step at a potential LP (*e.g.*, line 5 of readPair), we execute the abstract operation speculatively and keep both the result and the original abstract configuration. Later based on the result of the validation (*e.g.*, line 6 in readPair), we keep the appropriate branch and discard the other.

For the logic, we introduce two new auxiliary commands: **trylinself** is to do speculation, and **commit**(p) will commit to the appropriate branch satisfying the assertion p. In Fig. 1(c), we insert lines 5' and 6', where cid $\rightarrowtail$ $(\textbf{end}, (\texttt{a}, \texttt{b}))$ means that the current thread cid should have done its abstract operation and would return $(\texttt{a}, \texttt{b})$. We also extend the auxiliary state to record the multiple possibilities of abstract operations and abstract states after speculation.

Furthermore, we can combine the speculation idea with the pending thread pool. We allow the abstract operations in the pending thread pool as well as the current thread to speculate. Then we could handle some trickier algorithms such as RDCSS [12], in which the location of LP for thread t may be in the code of some other thread and also depend on the future behaviors of that thread. Please see Sec. 6 for one such example.

2.4 Simulation as Meta-Theory

The LP proof method can be understood as building simulations between the concrete implementations and the abstract atomic operations, such as the simple weak simulation in Fig. 2(a). The lower-level and higher-level arrows are the steps of the implementation and of the abstract operation respectively, and the dashed lines denote the simulation relation. We use dark nodes and white nodes at the abstract level to distinguish whether the operation has been finished or not. The only step at the concrete side corresponding to the single abstract step should be the LP of the implementation (labeled "LP" in the diagram). Since our program logic is based on the LP method, we can expect simulations to justify its soundness. In particular, we want a *thread-local* simulation which can handle both the helping mechanism and future-dependent LPs and can ensure linearizability.

To support helping in the simulation, we should allow the LP step at the concrete level to correspond to an abstract step made by a thread other than the one being verified. This requires information from other threads at the abstract side, thus makes it difficult to build a thread-local simulation. To address the problem, we introduce the pending thread pool at the abstract level of the simulation, just as in the development of our logic in Sec. 2.2. The new simula-

$$
\begin{array}{rlll}
(\textit{MName}) & f & \in & \textit{String} \\
(\textit{Expr}) & E & ::= & x \mid n \mid E+E \mid \ldots \\
(\textit{BExp}) & B & ::= & \textbf{true} \mid \textbf{false} \mid E=E \mid !B \mid \ldots \\
(\textit{Instr}) & c & ::= & x := E \mid x := [E] \mid [E] := E \mid \textbf{print}(E) \\
 & & & \mid x := \textbf{cons}(E,\ldots,E) \mid \textbf{dispose}(E) \mid \ldots \\
(\textit{Stmt}) & C & ::= & \textbf{skip} \mid c \mid x := f(E) \mid \textbf{return}\ E \mid \textbf{noret} \\
 & & & \mid \langle C \rangle \mid C;C \mid \textbf{if}\ (B)\ C\ \textbf{else}\ C \mid \textbf{while}\ (B)\{C\} \\
(\textit{Prog}) & W & ::= & \textbf{skip} \mid \textbf{let}\ \Pi\ \textbf{in}\ C \,\|\, \ldots \,\|\, C \\
(\textit{ODecl}) & \Pi & ::= & \{f_1 \rightsquigarrow (x_1, C_1), \ldots, f_n \rightsquigarrow (x_n, C_n)\}
\end{array}
$$

Figure 3. Syntax of the Programming Language

$$
\begin{array}{rlll}
(\textit{ThrdID}) & \mathtt{t} & \in & \textit{Nat} \\
(\textit{Mem}) & \sigma & \in & (\textit{PVar} \cup \textit{Nat}) \rightharpoonup \textit{Int} \\
(\textit{CallStk}) & \kappa & ::= & (\sigma_l, x, C) \mid \circ \\
(\textit{ThrdPool}) & \mathcal{K} & ::= & \{\mathtt{t}_1 \rightsquigarrow \kappa_1, \ldots, \mathtt{t}_n \rightsquigarrow \kappa_n\} \\
(\textit{PState}) & \mathcal{S} & ::= & (\sigma_c, \sigma_o, \mathcal{K}) \\
(\textit{LState}) & s & ::= & (\sigma_c, \sigma_o, \kappa) \\
(\textit{Evt}) & e & ::= & (\mathtt{t}, f, n) \mid (\mathtt{t}, \textbf{ok}, n) \mid (\mathtt{t}, \textbf{obj}, \textbf{abort}) \\
 & & & \mid (\mathtt{t}, \textbf{out}, n) \mid (\mathtt{t}, \textbf{clt}, \textbf{abort}) \\
(\textit{ETrace}) & H & ::= & \epsilon \mid e :: H
\end{array}
$$

Figure 4. States and Event Traces

tion is shown in Fig. 2(b). We can see that a concrete step of thread t could help linearize the operation of t′ in the pending thread pool as well as its own operation. Thus the new simulation intuitively supports the helping mechanism.

As forward simulations, neither of the simulations in Fig. 2(a) and (b) supports future-dependent LPs. For each step along the concrete execution in those simulations, we need to decide immediately whether the step is at the LP, and cannot postpone the decision to the future. As discussed a lot in previous work (*e.g.*, [1, 3, 6, 21]), we have to introduce backward simulations or hybrid simulations to support future-dependent LPs. Here we exploit the speculation idea and develop a forward-backward simulation [21]. As shown in Fig. 2(c), we keep both speculations after the potential LP, where the higher black nodes result from executing the abstract operation and the lower white nodes record the original abstract configuration. Then at the validation step we commit to the correct branch.

Finally, to ensure linearizability, the thread-local simulation has to be *compositional*. As a counterexample, we can construct a simple simulation (like the one in Fig. 2(a)) between the following implementation C and the abstract atomic increment operation γ, but C is not linearizable *w.r.t.* γ.

```
C : local t; t := x; x := t + 1;        γ : x++
```

The reason is that the simple simulation is not compositional *w.r.t.* parallel compositions. To address this problem, we proposed a compositional simulation RGSim [19] in previous work. The idea is to parameterize the simple simulation with the interference with the environment, in the form of rely/guarantee conditions (R and G) [17]. RGSim says, the concrete executions are simulated by the abstract executions under interference from the environment R, and all the related state transitions of the thread being verified should satisfy G. For parallel composition, we check that the guarantee G of each thread is permitted in the rely R of the other. Then the simulation becomes compositional and can ensure linearizability.

We combine the above ideas and develop a new compositional simulation with the support of non-fixed LPs as the meta-theory of our logic. We will discuss our simulation formally in Sec. 5.

3. Basic Technical Settings and Linearizability

In this section, we formalize linearizability of an object implementation *w.r.t.* its specification, and show that linearizability is equivalent to contextual refinement.

3.1 Language and Semantics

As shown in Fig. 3, a program W contains several client threads in parallel, each of which could call the methods declared in the object Π. A method is defined as a pair (x, C), where x is the formal argument and C is the method body. For simplicity, we assume there is only one object in W and each method takes one argument only, but it is easy to extend our work with multiple objects and arguments.

We use a runtime command **noret** to abort methods that terminate but do not execute **return** E. It is automatically appended to the method code and is not supposed to be used by programmers. Other commands are mostly standard. Clients can use **print**(E) to produce observable external events. We do not allow the object's methods to produce external events. To simplify the semantics, we also assume there are no nested method calls.

Figure 4 gives the model of program states. Here we partition a global state $\mathcal{S}$ into the client memory σ_c, the object σ_o and a thread pool $\mathcal{K}$. A client can only access the client memory σ_c, unless it calls object methods. The thread pool maps each thread id t to its local call stack frame. A call stack κ could be either empty ($\circ$) when the thread is not executing a method, or a triple (σ_l, x, C), where σ_l maps the method's formal argument and local variables (if any) to their values, x is the caller's variable to receive the return value, and C is the caller's remaining code to be executed after the method returns. To give a thread-local semantics, we also define the thread local view s of the state.

Figure 5 gives selected rules of the operational semantics. We show three kinds of transitions: $\longmapsto$ for the top-level program transitions, $\longrightarrow_{\mathtt{t},\Pi}$ for the transitions of thread t with the methods' declaration Π, and $\longrightarrow\!\!\!\!\!\triangleright_{\mathtt{t}}$ for the steps inside method calls of thread t. To describe the operational semantics for threads, we use an execution context **E**:

$$(\textit{ExecContext})\ \mathbf{E} ::= [\,] \mid \mathbf{E}; C$$

The hole $[\,]$ shows the place where the execution of code occurs. $\mathbf{E}[\,C\,]$ represents the code resulting from placing C into the hole.

We label transitions with events e defined in Fig. 4. An event could be a method invocation $(\mathtt{t}, f, n)$ or return $(\mathtt{t}, \textbf{ok}, n)$, a fault $(\mathtt{t}, \textbf{obj}, \textbf{abort})$ produced by the object method code, an output $(\mathtt{t}, \textbf{out}, n)$ generated by **print**(E), or a fault $(\mathtt{t}, \textbf{clt}, \textbf{abort})$ from the client code. The first two events are called object events, and the last two are observable external events. The third one $(\mathtt{t}, \textbf{obj}, \textbf{abort})$ belongs to both classes. An event trace H is then defined as a finite sequence of events.

3.2 Object Specification and Linearizability

Next we formalize the object specification Γ, which maps method names to their abstract operations γ, as shown in Fig. 6. γ transforms an argument value and an initial abstract object to a return value with a resulting abstract object in a single step. It specifies the intended sequential behaviors of the method. The abstract object representation θ is defined as a mapping from program variables to abstract values. We leave the abstract values unspecified here, which can be instantiated by programmers.

Then we give an abstract version of programs $\mathbb{W}$, where clients interact with the abstract object specification Γ instead of its implementation Π. The semantics is almost the same as the concrete language shown in Sec. 3.1, except that the abstract atomic operation γ is executed when the method is called, which now operates over the abstract object θ instead of over the concrete one σ_o. The

$$\frac{(C_i, (\sigma_c, \sigma_o, \mathcal{K}(i))) \xrightarrow{e}_{i,\Pi} (C'_i, (\sigma'_c, \sigma'_o, \kappa'))}{(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots C_i \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \mathcal{K})) \overset{e}{\longmapsto} (\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots C'_i \ldots \,\|\, C_n, (\sigma'_c, \sigma'_o, \mathcal{K}\{i \rightsquigarrow \kappa'\}))}$$

(a) Program Transitions

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

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

$$\frac{\kappa = (\sigma_l, x, C) \qquad [\![E]\!]_{\sigma_o \uplus \sigma_l} = n \qquad \sigma'_c = \sigma_c\{x \rightsquigarrow n\}}{(\mathbf{E}[\,\textbf{return}\ E\,], (\sigma_c, \sigma_o, \kappa)) \xrightarrow{(\mathsf{t}, \textbf{ok}, n)}_{\mathsf{t},\Pi} (C, (\sigma'_c, \sigma_o, \circ))}$$

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

$$\frac{}{(\textbf{noret}, s) \xrightarrow{(\mathsf{t}, \textbf{obj}, \textbf{abort})}_{\mathsf{t},\Pi} \textbf{abort}}$$

$$\frac{(C, \sigma_o \uplus \sigma_l) \mathrel{\longrightarrow\!\!\triangleright}_{\mathsf{t}} (C', \sigma'_o \uplus \sigma'_l) \qquad \mathit{dom}(\sigma_l) = \mathit{dom}(\sigma'_l)}{(C, (\sigma_c, \sigma_o, (\sigma_l, x, C_c))) \longrightarrow_{\mathsf{t},\Pi} (C', (\sigma_c, \sigma'_o, (\sigma'_l, x, C_c)))}$$

(b) Thread Transitions

$$\frac{[\![E_j]\!]_{\sigma}\ \mathit{undefined} \quad (1 \le j \le i) \quad \text{or} \quad x \notin \mathit{dom}(\sigma)}{(\mathbf{E}[\,x := \textbf{cons}(E_1, \ldots, E_i)\,], \sigma) \mathrel{\longrightarrow\!\!\triangleright}_{\mathsf{t}} \textbf{abort}}$$

$$\frac{(C, \sigma) \mathrel{\longrightarrow\!\!\triangleright}^{*}_{\mathsf{t}} (\textbf{skip}, \sigma')}{(\mathbf{E}[\,\langle C \rangle\,], \sigma) \mathrel{\longrightarrow\!\!\triangleright}_{\mathsf{t}} (\mathbf{E}[\,\textbf{skip}\,], \sigma')} \qquad \frac{(C, \sigma) \mathrel{\longrightarrow\!\!\triangleright}^{*}_{\mathsf{t}} \textbf{abort}}{(\mathbf{E}[\,\langle C \rangle\,], \sigma) \mathrel{\longrightarrow\!\!\triangleright}_{\mathsf{t}} \textbf{abort}}$$

(c) Thread Transitions Inside Method Calls

Figure 5. Selected Rules of Concrete Operational Semantics

$$\begin{array}{rrcl}
(\mathit{AbsObj}) & \theta & \in & \mathit{PVar} \rightharpoonup \mathit{AbsVal} \\
(\mathit{MSpec}) & \gamma & \in & \mathit{Int} \to \mathit{AbsObj} \rightharpoonup \mathit{Int} \times \mathit{AbsObj} \\
(\mathit{OSpec}) & \Gamma & ::= & \{f_1 \rightsquigarrow \gamma_1, \ldots, f_n \rightsquigarrow \gamma_n\} \\
(\mathit{AbsProg}) & \mathbb{W} & ::= & \textbf{skip} \ \mid\ \textbf{with}\ \Gamma\ \textbf{do}\ C \,\|\, \ldots \,\|\, C
\end{array}$$

Figure 6. Object Specification and Abstract Program

abstract operation generates a pair of invocation and return events atomically. Due to space limit, we give the semantics in TR [18].

Linearizability. Linearizability [16] is defined using the notion of histories, which are special event traces H consisting of only object events (*i.e.*, invocations, returns and object faults).

Below we use $H(i)$ for the i-th event of H, and $|H|$ for the length of H. $H|_{\mathsf{t}}$ represents the sub-history consisting of all the events whose thread id is t. The predicates $\mathsf{is\_inv}(e)$ and $\mathsf{is\_res}(e)$ mean that the event e is a method invocation and a response (*i.e.*, a return or an object fault) respectively. We say a response e_2 *matches* an invocation e_1 iff they have the same thread id.

A history H is *sequential* iff the first event of H is an invocation, and each invocation, except possibly the last, is immediately followed by a matching response. Then H is *well-formed* iff, for all t, $H|_{\mathsf{t}}$ is sequential. H is *complete* iff it is well-formed and every invocation has a matching response. An invocation is *pending* if no matching response follows it. We handle pending invocations in an incomplete history H following the standard linearizability definition [16]: we append zero or more response events to H, and drop the remaining pending invocations. Then we get a set of complete histories, which is denoted by $\mathsf{completions}(H)$. We give formal definitions of the above concepts in TR [18].

Definition 1 (Linearizable Histories). $H \preceq_{\mathsf{lin}} H'$ iff

1. $\forall \mathsf{t}.\ H|_{\mathsf{t}} = H'|_{\mathsf{t}}$;
2. there exists a bijection $\pi : \{1, \ldots, |H|\} \to \{1, \ldots, |H'|\}$ such that $\forall i.\ H(i) = H'(\pi(i))$ and

   $\forall i, j.\ i < j \wedge \mathsf{is\_res}(H(i)) \wedge \mathsf{is\_inv}(H(j)) \implies \pi(i) < \pi(j)$.

That is, H is linearizable *w.r.t.* H' 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.

We use $\mathcal{H}[\![W, (\sigma_c, \sigma_o)]\!]$ to represent the set of histories produced by the executions of W with the initial client memory σ_c, the object σ_o, and empty call stacks for all threads, and use $\mathcal{H}[\![\mathbb{W}, (\sigma_c, \theta)]\!]$ to generate histories from the abstract program $\mathbb{W}$ with the initial client memory σ_c and the abstract object θ.

A *legal* sequential history H is a history generated by any client using the specification Γ and an initial abstract object θ.

$$\begin{array}{l}
\Gamma \triangleright (\theta, H) \ \stackrel{\text{def}}{=} \\
\exists n, C_1, \ldots, C_n, \sigma_c.\ H \in \mathcal{H}[\![(\textbf{with}\ \Gamma\ \textbf{do}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \theta)]\!]
\end{array}$$

Then an *object* is linearizable iff all its completed concurrent histories are linearizable *w.r.t.* some legal sequential histories.

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

$$\begin{array}{l}
\forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, \theta, H. \\
H \in \mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o)]\!] \wedge (\varphi(\sigma_o) = \theta) \\
\implies \exists H_c, H'.\ H_c \in \mathsf{completions}(H) \wedge \Gamma \triangleright (\theta, H') \wedge H_c \preceq_{\mathsf{lin}} H'
\end{array}$$

Here the mapping φ relates concrete objects to abstract ones:

$$(\mathit{RefMap}) \ \ \varphi \ \in \ \mathit{Mem} \rightharpoonup \mathit{AbsObj}$$

The side condition $\varphi(\sigma_o) = \theta$ in the above definition requires the initial concrete object σ_o to be a well-formed data structure representing a valid object θ.

3.3 Contextual Refinement and Linearizability

Next we define contextual refinement between the concrete object and its specification, and prove its equivalence to linearizability. This equivalence will serve as the basis of our logic soundness *w.r.t.* linearizability.

Informally, the contextual refinement says, for any set of client threads, the program W has no more observable behaviors than the corresponding abstract program $\mathbb{W}$. Below we use $\mathcal{O}[\![W, (\sigma_c, \sigma_o)]\!]$ to represent the set of observable event traces generated during the executions of W with the initial state (σ_c, σ_o) (and empty stacks). It is defined similarly as $\mathcal{H}[\![W, (\sigma_c, \sigma_o)]\!]$, but now the traces consist of observable events only (output events, client faults or object faults). The observable event traces $\mathcal{O}[\![\mathbb{W}, (\sigma_c, \theta)]\!]$ generated by the abstract program is defined similarly.

$$
\begin{array}{rrcl}
(\mathit{InsStmt}) & \widetilde{C} & ::= & \textbf{skip} \mid c \mid \textbf{return}\, E \mid \textbf{noret} \\
 & & \mid & \textbf{linself} \mid \textbf{lin}(E) \mid \textbf{trylinself} \\
 & & \mid & \textbf{trylin}(E) \mid \textbf{commit}(p) \mid \langle \widetilde{C} \rangle \mid \widetilde{C}; \widetilde{C} \\
 & & \mid & \textbf{if}\,(B)\, \widetilde{C}\, \textbf{else}\, \widetilde{C} \mid \textbf{while}\,(B)\{\widetilde{C}\} \\
(\mathit{RelState}) & \Sigma & ::= & (\sigma, \Delta) \\
(\mathit{SpecSet}) & \Delta & ::= & \{(U_1, \theta_1), \ldots, (U_n, \theta_n)\} \\
(\mathit{PendThrds}) & U & ::= & \{\mathsf{t}_1 \leadsto \Upsilon_1, \ldots, \mathsf{t}_n \leadsto \Upsilon_n\} \\
(\mathit{AbsOp}) & \Upsilon & ::= & (\gamma, n) \mid (\textbf{end}, n) \\
(\mathit{RelAss}) & p, q, I & ::= & \mathsf{true} \mid \mathsf{false} \mid E = E \mid \mathsf{emp} \mid E \mapsto E \\
 & & \mid & x \Mapsto E \mid E \rightarrowtail (\gamma, E) \mid E \rightarrowtail (\textbf{end}, E) \\
 & & \mid & p * q \mid p \oplus q \mid p \vee q \mid \ldots \\
(\mathit{RelAct}) & R, G & ::= & p \ltimes q \mid [p] \mid R * R \mid R \oplus R \mid \ldots
\end{array}
$$

Figure 7. Instrumented Code and Relational State Model

$$
\bullet \overset{\text{def}}{=} \{(\emptyset, \emptyset)\} \qquad \text{where } \bullet \in \mathit{SpecSet}
$$

$$
\begin{array}{l}
f \perp g \overset{\text{def}}{=} \mathit{dom}(f) \cap \mathit{dom}(g) = \emptyset \\
\Delta_1 \sharp \Delta_2 \overset{\text{def}}{=} U_1 \perp U_2 \wedge \theta_1 \perp \theta_2\,, \text{ where } (U_1, \theta_1) \in \Delta_1 \wedge (U_2, \theta_2) \in \Delta_2 \\
\Delta_1 * \Delta_2 \overset{\text{def}}{=} \{(U_1 \uplus U_2, \theta_1 \uplus \theta_2) \mid (U_1, \theta_1) \in \Delta_1 \wedge (U_2, \theta_2) \in \Delta_2\} \\
\Sigma_1 * \Sigma_2 \overset{\text{def}}{=} (\sigma_1 \uplus \sigma_2, \Delta_1 * \Delta_2) \\
\qquad \text{where } \Sigma_1 = (\sigma_1, \Delta_1), \Sigma_2 = (\sigma_2, \Delta_2), \sigma_1 \perp \sigma_2 \text{ and } \Delta_1 \sharp \Delta_2
\end{array}
$$

$$
\Sigma_1 \oplus \Sigma_2 \overset{\text{def}}{=} \begin{cases} (\sigma, \Delta_1 \cup \Delta_2) & \text{if } \Sigma_1 = (\sigma, \Delta_1) \text{ and } \Sigma_2 = (\sigma, \Delta_2) \\ \mathit{undefined} & \text{otherwise} \end{cases}
$$

$$
\{\!\!\{E\}\!\!\}_\sigma \overset{\text{def}}{=} \begin{cases} [\![E]\!]_\sigma & \text{if } \mathit{dom}(\sigma) = \mathit{fv}(E) \\ \mathit{undefined} & \text{otherwise} \end{cases}
$$

$$
\begin{array}{lll}
(\sigma, \Delta) \models E_1 = E_2 & \text{iff} & \{\!\!\{(E_1 = E_2)\}\!\!\}_\sigma = \textbf{true} \wedge \Delta = \bullet \\
(\sigma, \Delta) \models \mathsf{emp} & \text{iff} & \sigma = \emptyset \wedge \Delta = \bullet \\
(\sigma, \Delta) \models E_1 \mapsto E_2 & \text{iff} & \exists l, n, \sigma'.\ \{\!\!\{(E_1, E_2)\}\!\!\}_{\sigma'} = (l, n) \\
 & & \quad \wedge\, \sigma = \sigma' \uplus \{l \leadsto n\} \wedge \Delta = \bullet \\
(\sigma, \Delta) \models x \Mapsto E & \text{iff} & \exists n, \theta.\ \{\!\!\{E\}\!\!\}_\sigma = n \wedge \theta = \{x \leadsto n\} \\
 & & \quad \wedge\, \Delta = \{(\emptyset, \theta)\} \\
(\sigma, \Delta) \models E_1 \rightarrowtail (\gamma, E_2) & \text{iff} & \exists \sigma_1, \sigma_2, \mathsf{t}, n.\ \sigma = \sigma_1 \uplus \sigma_2 \\
 & & \quad \wedge \{\!\!\{E_1\}\!\!\}_{\sigma_1} = \mathsf{t} \wedge \{\!\!\{E_2\}\!\!\}_{\sigma_2} = n \\
 & & \quad \wedge\, \Delta = \{(\{\mathsf{t} \leadsto (\gamma, n)\}, \emptyset)\} \\
(\sigma, \Delta) \models E_1 \rightarrowtail (\textbf{end}, E_2) & \text{iff} & \exists \sigma_1, \sigma_2, \mathsf{t}, n.\ \sigma = \sigma_1 \uplus \sigma_2 \\
 & & \quad \wedge \{\!\!\{E_1\}\!\!\}_{\sigma_1} = \mathsf{t} \wedge \{\!\!\{E_2\}\!\!\}_{\sigma_2} = n \\
 & & \quad \wedge\, \Delta = \{(\{\mathsf{t} \leadsto (\textbf{end}, n)\}, \emptyset)\}
\end{array}
$$

$$
\begin{array}{l}
\Sigma \models p * q \ \text{ iff } \ \exists \Sigma_1, \Sigma_2.\ \Sigma = \Sigma_1 * \Sigma_2 \wedge \Sigma_1 \models p \wedge \Sigma_2 \models q \\
\Sigma \models p \oplus q \ \text{ iff } \ \exists \Sigma_1, \Sigma_2.\ \Sigma = \Sigma_1 \oplus \Sigma_2 \wedge \Sigma_1 \models p \wedge \Sigma_2 \models q \\
\mathsf{SpecExact}(p) \ \text{ iff } \ \forall \Delta, \Delta'.\ ((\_, \Delta) \models p) \wedge ((\_, \Delta') \models p) \Longrightarrow (\Delta = \Delta')
\end{array}
$$

Figure 8. Semantics of State Assertions

Definition 3 (Contextual Refinement). $\Pi \sqsubseteq_\varphi \Gamma$ iff

$$
\begin{array}{l}
\forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, \theta.\ \varphi(\sigma_o) = \theta \\
\quad \Longrightarrow \mathcal{O}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma_o)]\!] \\
\qquad \subseteq \mathcal{O}[\![(\textbf{with}\ \Gamma\ \textbf{do}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \theta)]\!].
\end{array}
$$

Following Filipović *et al.* [9], we can prove that linearizability is equivalent to contextual refinement. We give the proofs in TR [18].

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

4. A Relational Rely-Guarantee Style Logic

To prove object linearizability, we first instrument the object implementation by introducing auxiliary states and auxiliary commands, which relate the concrete code with the abstract object and operations. Our program logic extends LRG [8] with a relational interpretation of assertions and new rules for auxiliary commands.

$$
\begin{array}{l}
(\Sigma, \Sigma') \models p \ltimes q \ \text{ iff } \ \Sigma \models p \wedge \Sigma' \models q \\
(\Sigma, \Sigma') \models [p] \ \text{ iff } \ \Sigma \models p \wedge \Sigma = \Sigma' \\
(\Sigma, \Sigma') \models R_1 * R_2 \ \text{ iff } \\
\qquad \exists \Sigma_1, \Sigma_2, \Sigma_1', \Sigma_2'.\ (\Sigma = \Sigma_1 * \Sigma_2) \wedge (\Sigma' = \Sigma_1' * \Sigma_2') \\
\qquad\qquad \wedge (\Sigma_1, \Sigma_1') \models R_1 \wedge (\Sigma_2, \Sigma_2') \models R_2 \\
(\Sigma, \Sigma') \models R_1 \oplus R_2 \ \text{ iff } \\
\qquad \exists \Sigma_1, \Sigma_2, \Sigma_1', \Sigma_2'.\ (\Sigma = \Sigma_1 \oplus \Sigma_2) \wedge (\Sigma' = \Sigma_1' \oplus \Sigma_2') \\
\qquad\qquad \wedge (\Sigma_1, \Sigma_1') \models R_1 \wedge (\Sigma_2, \Sigma_2') \models R_2
\end{array}
$$

$$
\mathsf{Id} \overset{\text{def}}{=} [\mathsf{true}] \qquad\qquad \mathsf{True} \overset{\text{def}}{=} \mathsf{true} \ltimes \mathsf{true}
$$

$$
\frac{}{(\emptyset, n) \xrightarrow{\gamma} (\emptyset, n')} \qquad \frac{\gamma(n)(\theta) = (n', \theta') \quad (\Delta, n) \xrightarrow{\gamma} (\Delta', n')}{(\{(U, \theta)\} \uplus \Delta, n) \xrightarrow{\gamma} (\{(U, \theta')\} \uplus \Delta', n')}
$$

$$
\begin{array}{l}
[E, p]\gamma[E', q] \ \text{ iff } \\
\qquad \forall \sigma, \Delta, n.\ (\sigma, \Delta) \models (E = n) * p \\
\qquad \Longrightarrow \ \exists \Delta', n'.\ (\Delta, n) \xrightarrow{\gamma} (\Delta', n') \wedge ((\sigma, \Delta') \models (E' = n') * q)
\end{array}
$$

$$
I \triangleright R \ \text{ iff } \ ([I] \Rightarrow R) \wedge (R \Rightarrow I \ltimes I) \wedge \mathsf{Precise}(I)
$$

Figure 9. Semantics of Actions

Although our logic is based on LRG [8], this approach is mostly independent with the base logic. Similar extensions can also be made over other logics, such as RGSep [32].

Our logic is proposed to verify object methods only. Verified object methods are guaranteed to be a contextual refinement of their abstract atomic operations, which ensures linearizability of the object. We discuss verification of whole programs consisting of both client code and object code at the end of Sec. 4.3.

4.1 Instrumented Code and States

In Fig. 7 we show the syntax of the instrumented code and its state model. As explained in Sec. 2, program states Σ for the object method executions now consist of two parts, the physical object states σ and the auxiliary data Δ. Δ is a *nonempty* set of (U, θ) pairs, each pair representing a speculation of the situation at the abstract level. Here θ is the current abstract object, and U is a pending thread pool recording the remaining operation to be fulfilled by each thread. It maps a thread id to its remaining abstract operation, which is either (γ, n) (the operation γ needs to be executed with argument n) or $(\textbf{end}, n)$ (the operation has been finished with the return value n). We assume Δ is always *domain-exact*, defined as follows:

$$
\begin{array}{l}
\mathsf{DomExact}(\Delta) \overset{\text{def}}{=} \forall U, \theta, U', \theta'.\ (U, \theta) \in \Delta \wedge (U', \theta') \in \Delta \\
\qquad\qquad \Longrightarrow \mathit{dom}(U) = \mathit{dom}(U') \wedge \mathit{dom}(\theta) = \mathit{dom}(\theta')\,.
\end{array}
$$

It says, all the speculations in Δ should describe the same set of threads and the same domain of abstract objects. Any Δ containing a single speculation is domain-exact. Also domain-exactness can be preserved under the step of any command in our instrumented language, thus it is reasonable to assume it always holds.

Below we informally explain the effects over Δ of the newly introduced commands. We leave their formal semantics to Sec. 4.4. The auxiliary command **linself** executes the unfinished abstract operation of the current thread in every U in Δ, and changes the abstract object θ correspondingly. **lin**(E) executes the abstract operation of the thread with id E. **linself** or **lin**(E) is executed when we know for sure that a step is the linearization point. The **trylinself** command introduces uncertainty. Since we do not know if the abstract operation of the current thread is fulfilled or not at the current point, we consider both possibilities. For each (U, θ) pair in Δ that contains unfinished abstract operation of the current thread, we add in Δ a new speculation (U', θ') where the abstract operation is done and θ' is the resulting abstract object. Since the original (U, θ) is also kept, we have both speculations in Δ. Similarly, the

$$\frac{[E_1,p]\gamma[E_2,q]}{\vdash_{\mathsf{t}} \{\mathsf{t} \rightarrowtail (\gamma, E_1) * p\}\textbf{linself}\{\mathsf{t} \rightarrowtail (\textbf{end}, E_2) * q\}}\;(\textsc{linself}) \qquad \frac{}{\vdash_{\mathsf{t}} \{\mathsf{t} \rightarrowtail (\textbf{end}, E)\}\textbf{linself}\{\mathsf{t} \rightarrowtail (\textbf{end}, E)\}}\;(\textsc{linself-end})$$

$$\frac{[E_1,p]\gamma[E_2,q]}{\vdash_{\mathsf{t}} \{E \rightarrowtail (\gamma, E_1) * p\}\textbf{trylin}(E)\{(E \rightarrowtail (\gamma, E_1) * p) \oplus (E \rightarrowtail (\textbf{end}, E_2) * q)\}}\;(\textsc{try})$$

$$\frac{}{\vdash_{\mathsf{t}} \{E \rightarrowtail (\textbf{end}, E')\}\textbf{trylin}(E)\{E \rightarrowtail (\textbf{end}, E')\}}\;(\textsc{try-end}) \qquad \frac{\mathsf{SpecExact}(p) \qquad p' \Rightarrow p}{\vdash_{\mathsf{t}} \{p' \oplus \mathsf{true}\}\textbf{commit}(p)\{p'\}}\;(\textsc{commit})$$

$$\frac{}{\vdash_{\mathsf{t}} \{\mathsf{t} \rightarrowtail (\textbf{end}, E)\}\mathbf{E}[\,\textbf{return}\ E\,]\{\mathsf{t} \rightarrowtail (\textbf{end}, E)\}}\;(\textsc{ret}) \qquad \frac{\vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\}}{\vdash_{\mathsf{t}} \{p * r\}\widetilde{C}\{q * r\}}\;(\textsc{frame}) \qquad \frac{\vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\} \qquad \vdash_{\mathsf{t}} \{p'\}\widetilde{C}\{q'\}}{\vdash_{\mathsf{t}} \{p \oplus p'\}\widetilde{C}\{q \oplus q'\}}\;(\textsc{Spec-Conj})$$

$$\frac{\begin{array}{ll}\vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\} & (p \ltimes q) \Rightarrow G * \mathsf{True} \\ I \triangleright G & p \vee q \Rightarrow I * \mathsf{true}\end{array}}{[I], G, I \vdash_{\mathsf{t}} \{p\}\langle\widetilde{C}\rangle\{q\}}\;(\textsc{atom}) \qquad \frac{\begin{array}{c}[I], G, I \vdash_{\mathsf{t}} \{p\}\langle\widetilde{C}\rangle\{q\} \\ \mathsf{Sta}(\{p,q\}, R * \mathsf{Id}) \quad I \triangleright R\end{array}}{R, G, I \vdash_{\mathsf{t}} \{p\}\langle\widetilde{C}\rangle\{q\}}\;(\textsc{atom-r})$$

Figure 10. Selected Inference Rules

trylin(E) command introduces speculations about the thread E. When we have enough knowledge p about the situation of the abstract objects and operations, the **commit**(p) step keeps only the subset of speculations consistent with p and drops the rest. Here p is a logical assertion about the state Σ, which is explained below.

4.2 Assertions

Syntax of assertions is shown in Fig. 7. Following rely-guarantee style reasoning, assertions are either single state assertions p and q or binary rely/guarantee conditions R and G. Note here states refer to the relational states Σ.

We use standard separation logic assertions such as true, $E_1 = E_2$, emp and $E_1 \mapsto E_2$ to specify the memory σ. As shown in Fig. 8, their semantics is almost standard, but for $E_1 = E_2$ to hold over σ we require the domain of σ contains only the free variables in E_1 and E_2. Here we use $\{\!\!\{E\}\!\!\}_\sigma$ to evaluate E with the extra requirement that σ contains the exact resource to do the evaluation.

New assertions are introduced to specify Δ. $x \Mapsto E$ specifies the abstract object θ in Δ, with no speculations of U (abstract operations), while $E_1 \rightarrowtail (\gamma, E_2)$ (and $E_1 \rightarrowtail (\textbf{end}, E_2)$) specifies the singleton speculation of U. Semantics of separating conjunction $p * q$ is similar as in separation logic, except that it is now lifted to assertions over the relational states Σ. Note that the underlying "disjoint union" over Δ for separating conjunction should not be confused with the normal disjoint union operator over sets. The former (denoted as $\Delta_1 * \Delta_2$ in Fig. 8) describes the split of pending thread pools and/or abstract objects. For example, the left side Δ in the following equation specifies two speculations of threads t_1 and t_2 (we assume the abstract object part is empty and omitted here), and it can be split into two sets Δ_1 and Δ_2 on the right side, each of which describes the speculations of a single thread.

$$\left\{ \begin{array}{ll}\mathsf{t}_1 & \boxed{\Upsilon_1} \\ \mathsf{t}_2 & \boxed{\Upsilon_2}\end{array},\ \begin{array}{ll}\mathsf{t}_1 & \boxed{\Upsilon_1} \\ \mathsf{t}_2 & \boxed{\Upsilon'_2}\end{array} \right\} = \begin{array}{c}\{\mathsf{t}_1\ \boxed{\Upsilon_1}\} \\ * \\ \{\mathsf{t}_2\ \boxed{\Upsilon_2},\ \ \mathsf{t}_2\ \boxed{\Upsilon'_2}\}\end{array}$$

The most interesting new assertion is $p \oplus q$, where p and q specify two different speculations. It is this assertion that reflects uncertainty about the abstract level. However, the readers should not confuse $\oplus$ with disjunction. It is more like conjunction since it says Δ contains both speculations satisfying p *and* those satisfying q. As an example, the above equation could be formulated at the assertion level using $*$ and $\oplus$:

$$\begin{array}{rl} & (\mathsf{t}_1 \rightarrowtail \Upsilon_1 * \mathsf{t}_2 \rightarrowtail \Upsilon_2) \oplus (\mathsf{t}_1 \rightarrowtail \Upsilon_1 * \mathsf{t}_2 \rightarrowtail \Upsilon'_2) \\ \Leftrightarrow & \mathsf{t}_1 \rightarrowtail \Upsilon_1 * (\mathsf{t}_2 \rightarrowtail \Upsilon_2 \oplus \mathsf{t}_2 \rightarrowtail \Upsilon'_2)\end{array}$$

Rely and guarantee assertions specify transitions over Σ. Here we follow the syntax of LRG [8], with a new assertion $R_1 \oplus R_2$ specifying speculative behaviors of the environment. The semantics is given in Fig. 9. We will show the use of the assertions in the examples of Sec. 6.

4.3 Inference Rules

The rules of our logic are shown in Fig. 10. Rules on the top half are for sequential Hoare-style reasoning. They are proposed to verify code $\widetilde{C}$ in the atomic block $\langle\widetilde{C}\rangle$. The judgment is parameterized with the id t of the current thread.

For the **linself** command, if the abstract operation γ of the current thread has not been done, this command will finish it. Here $[E_1, p]\gamma[E_2, q]$ in the LINSELF rule describes the behavior of γ, which transforms abstract objects satisfying p to new ones satisfying q. E_1 and E_2 are the argument and return value respectively. The definition is given in Fig. 9. The LINSELF-END rule says **linself** has no effects if we know the abstract operation has been finished. The LIN rule and LIN-END rule are similar and omitted here.

The TRY rule says that if the thread E has not finished the abstract operation γ, it can do speculation using **trylin**(E). The resulting state contains both cases, one says γ does not progress at this point and the other says it does. If the current thread has already finished the abstract operation, **trylin**(E) would have no effects, as shown in the TRY-END rule. We omit the TRYSELF rule and TRYSELF-END rule for the current thread, which are similar.

The above two pairs of rules require us to know for sure either the abstract operation has been finished or not. If we want to support uncertainty in the pre-condition, we could first consider different cases and then apply the SPEC-CONJ rule, which is like the conjunction rule in traditional Hoare logic.

The COMMIT rule allows us to commit to a specific speculation and drop the rest. **commit**(p) keeps only the speculations satisfying p. We require p to describe an exact set of speculations, as defined by $\mathsf{SpecExact}(p)$ in Fig. 8. For example, the following p_1 is speculation-exact, while p_2 is not:

$$\begin{array}{rl} p_1 \stackrel{\text{def}}{=} & \mathsf{t} \rightarrowtail (\gamma, n) \oplus \mathsf{t} \rightarrowtail (\textbf{end}, n') \\ p_2 \stackrel{\text{def}}{=} & \mathsf{t} \rightarrowtail (\gamma, n) \vee \mathsf{t} \rightarrowtail (\textbf{end}, n')\end{array}$$

In all of our examples in Sec. 6, the assertion p in **commit**(p) describes a singleton speculation, so $\mathsf{SpecExact}(p)$ trivially holds.

Before the current thread returns, it must know its abstract operation has been done, as required in the RET rule. We also have a standard FRAME rule as in separation logic for local reasoning.

Rules in the bottom half show how to do rely-guarantee style concurrency reasoning, which are very similar to those in LRG [8]. As in LRG, we use a precise invariant I to specify the boundary of the well-formed shared resource. The ATOM rule says we could reason sequentially about code in the atomic block. Then we can lift it to the concurrent setting as long as its effects over the shared resource satisfy the guarantee G, which is fenced by the invariant I. In this step we assume the environment does not update shared resource, thus using Id as the rely condition (see Fig. 9). To allow general environment behaviors, we should apply the ATOM-R rule later, which requires that R be fenced by I and the pre- and post-conditions be stable with respect to R. Here $\mathsf{Sta}(\{p, q\}, R)$ requires that p and q be stable with respect to R, a standard requirement in rely-guarantee reasoning. More rules are shown in TR [18].

Linking with client program verification. Our relational logic is introduced for object verification, but it can also be used to verify client code, since it is just an extension over the general-purpose concurrent logic LRG (which includes the rule for parallel composition). Moreover, as we will see in Sec. 5, our logic ensures contextual refinement. Therefore, to verify a program W, we could replace the object implementation with the abstract operations and verify the corresponding abstract program $\mathbb{W}$ instead. Since $\mathbb{W}$ abstracts away concrete object representation and method implementation details, this approach provides us with "separation and information hiding" [26] over the object, but still keeps enough information (*i.e.*, the abstract operations) about the method calls in concurrent client verification.

4.4 Semantics and Partial Correctness

We first show some key operational semantics rules for the instrumented code in Fig. 11.

A single step execution of the instrumented code by thread t is represented as $(\widetilde{C}, \Sigma) \hookrightarrow_{\mathsf{t}} (\widetilde{C}', \Sigma')$. When we reach the **return** E command (the second rule), we require that there be no uncertainty about thread t at the abstract level in Δ. That is, in every speculation in Δ, we always know t's operation has been finished with the same return value E. Meanings of the auxiliary commands have been explained before. Here we use the auxiliary definition $\Delta \rightarrow_{\mathsf{t}} \Delta'$ to formally define their transitions over Δ. The semantics of **commit**(p) requires p to be speculation-exact (see Fig. 8). Also it uses $(\sigma, \Delta)|_p = (\sigma', \Delta')$ to filter out the wrong speculations. To ensure locality, this filter allows Δ to contain some extra resource such as the threads and their abstract operations other than those described in p. For example, the following Δ describes two threads t_1 and t_2, but we could mention only t_1 in **commit**(p).

$$\Delta : \left\{ \begin{array}{ll} \mathsf{t}_1 & \boxed{(\gamma_1, n_1)} \\ \mathsf{t}_2 & \boxed{(\gamma_2, n_2)} \end{array}, \quad \begin{array}{ll} \mathsf{t}_1 & \boxed{(\mathbf{end}, n_1')} \\ \mathsf{t}_2 & \boxed{(\mathbf{end}, n_2')} \end{array} \right\}$$

If p is $\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1)$, then **commit**$(p)$ will keep only the left speculation and discard the other. p can also be $\mathsf{t}_1 \rightarrowtail (\gamma_1, n_1) \oplus \mathsf{t}_1 \rightarrowtail (\mathbf{end}, n_1')$, then **commit**$(p)$ will keep both speculations.

Given the thread-local semantics, we could next define the transition $(\widetilde{C}, \Sigma) \overset{R}{\hookrightarrow}_{\mathsf{t}} (\widetilde{C}, \Sigma)$, which describes the behavior of thread t with interference R from the environment.

Semantics preservation by the instrumentation. It is easy to see that the newly introduced auxiliary commands do not change the physical state σ, nor do they affect the program control flow. Thus the instrumentation does not change program behaviors, unless the auxiliary commands are inserted into the wrong places and they get stuck, but this can be prevented by our program logic.

Soundness w.r.t. partial correctness. Following LRG [8], we could give semantics of the logic judgment as $R, G, I \models_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$, which encodes partial correctness of $\widetilde{C}$ *w.r.t.* the pre- and post-conditions. We could prove the logic ensures partial correctness by showing $R, G, I \vdash_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$ implies $R, G, I \models_{\mathsf{t}} \{p\}\widetilde{C}\{q\}$. The details are shown in TR [18]. In the next section, we give a stronger soundness of the logic, *i.e.* soundness *w.r.t.* linearizability.

$$\frac{(C, \sigma) \longrightarrow_{\mathsf{t}} (C', \sigma') \qquad C \neq \mathbf{E}[\,\mathbf{return}\ \_\,]}{(C, (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (C', (\sigma', \Delta))}$$

$$\frac{\forall U.\ (U, \_) \in \Delta \implies U(\mathsf{t}) = (\mathbf{end}, [\![E]\!]_\sigma)}{(\mathbf{E}[\,\mathbf{return}\ E\,], (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{skip}, (\sigma, \Delta))}$$

$$\frac{\Delta \rightarrow_{\mathsf{t}} \Delta'}{(\mathbf{E}[\,\mathbf{linself}\,], (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{E}[\,\mathbf{skip}\,], (\sigma, \Delta'))}$$

$$\frac{[\![E]\!]_\sigma = \mathsf{t}' \qquad \Delta \rightarrow_{\mathsf{t}'} \Delta'}{(\mathbf{E}[\,\mathbf{trylin}(E)\,], (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{E}[\,\mathbf{skip}\,], (\sigma, \Delta \cup \Delta'))}$$

$$\frac{\mathsf{SpecExact}(p) \qquad (\sigma, \Delta)|_p = (\_, \Delta')}{(\mathbf{E}[\,\mathbf{commit}(p)\,], (\sigma, \Delta)) \hookrightarrow_{\mathsf{t}} (\mathbf{E}[\,\mathbf{skip}\,], (\sigma, \Delta'))}$$

$$\frac{(\widetilde{C}, \Sigma) \hookrightarrow_{\mathsf{t}} (\widetilde{C}', \Sigma')}{(\widetilde{C}, \Sigma) \overset{R}{\hookrightarrow}_{\mathsf{t}} (\widetilde{C}', \Sigma')} \qquad\qquad \frac{(\Sigma, \Sigma') \models R}{(\widetilde{C}, \Sigma) \overset{R}{\hookrightarrow}_{\mathsf{t}} (\widetilde{C}, \Sigma')}$$

Auxiliary Definitions:

$$\frac{U(\mathsf{t}) = (\gamma, n) \qquad \gamma(n)(\theta) = (n', \theta')}{(U, \theta) \dashrightarrow_{\mathsf{t}} (U\{\mathsf{t} \rightsquigarrow (\mathbf{end}, n')\}, \theta')} \qquad\qquad \frac{U(\mathsf{t}) = (\mathbf{end}, n)}{(U, \theta) \dashrightarrow_{\mathsf{t}} (U, \theta)}$$

$$\frac{}{\emptyset \rightarrow_{\mathsf{t}} \emptyset} \qquad\qquad \frac{(U, \theta) \dashrightarrow_{\mathsf{t}} (U', \theta') \qquad \Delta \rightarrow_{\mathsf{t}} \Delta'}{\{(U, \theta)\} \uplus \Delta \ \rightarrow_{\mathsf{t}}\ \{(U', \theta')\} \cup \Delta'}$$

$$\begin{array}{l} (\sigma, \Delta)|_p = (\sigma', \Delta') \ \text{ iff} \\ \quad \exists \sigma'', \Delta'', \Delta_p.\ (\sigma = \sigma' \uplus \sigma'') \wedge (\Delta = \Delta' \uplus \Delta'') \wedge ((\sigma', \Delta_p) \models p) \\ \qquad\qquad\qquad \wedge (\Delta'|_{dom(\Delta_p)} = \Delta_p) \wedge (\Delta''|_{dom(\Delta_p)} \cap \Delta_p = \emptyset) \\ \Delta|_D \overset{\text{def}}{=} \{(U,\theta) \mid dom(\{(U,\theta)\}) = D \wedge \exists U', \theta'.\ (U \uplus U', \theta \uplus \theta') \in \Delta\} \\ dom(\Delta) \overset{\text{def}}{=} (dom(U), dom(\theta)) \qquad \text{where } (U, \theta) \in \Delta \end{array}$$

Figure 11. Operational Semantics in the Relational State Model

5. Soundness via Simulation

Our logic intuitively relates the concrete object code with its abstract level specification. In this section we formalize the intuition and prove that the logic indeed ensures object linearizability. The proof is constructed in the following steps. We propose a new rely-guarantee-based forward-backward simulation between the concrete code and the abstract operation. We prove the simulation is compositional and implies contextual refinement between the two sides, and our logic indeed establishes such a simulation. Thus the logic establishes contextual refinement. Finally we get linearizability following Theorem 4.

Below we first define a rely-guarantee-based forward-backward simulation. It extends RGSim [19] with the support of the helping mechanism and speculations.

Definition 5 (Simulation for Method). $(x, C) \preceq^{\mathsf{t}}_{R;G;p} \gamma$ iff

$$\begin{array}{l} \forall n, \sigma, \Delta.\ (\sigma, \Delta) \models (\mathsf{t} \rightarrowtail (\gamma, n) * (x = n) * p) \\ \qquad \implies (C; \mathbf{noret}, \sigma) \preceq^{\mathsf{t}}_{R;G;p} \Delta\,. \end{array}$$

Whenever $(C, \sigma) \preceq^{\mathsf{t}}_{R;G;p} \Delta$, we have the following:

1. if $C \neq \mathbf{E}[\,\mathbf{return}\ \_\,]$, then
   (a) for any C' and σ', if $(C, \sigma) \longrightarrow_{\mathsf{t}} (C', \sigma')$, then there exists Δ' such that $\Delta \rightrightarrows \Delta'$, $((\sigma, \Delta), (\sigma', \Delta')) \models (G * \mathsf{True})$ and $(C', \sigma') \preceq^{\mathsf{t}}_{R;G;p} \Delta'$;

(b) $(C, \sigma) \not\longrightarrow_{\mathsf{t}} \textbf{abort}$;

2. for any σ' and Δ', if $((\sigma, \Delta), (\sigma', \Delta')) \models (R * \mathsf{Id})$, then $(C, \sigma') \preceq^{\mathsf{t}}_{R;G;p} \Delta'$;
3. if $C = \mathbf{E}[\,\textbf{return}\ E\,]$, then there exists n' such that $[\![E]\!]_\sigma = n'$ and $(\sigma, \Delta) \models (\mathsf{t} \rightarrowtail (\textbf{end}, n') * (x = \_) * p)$.

As in RGSim, $(x, C) \preceq^{\mathsf{t}}_{R;G;p} \gamma$ says, the implementation C is simulated by the abstract operation γ under the interference with the environment, which is specified by R and G. The new simulation holds if the executions of the concrete code C are related to the *speculative* executions of some Δ. The Δ could specify abstract operations of other threads that might be helped, as well as the current thread t. Initially, the abstract operation of t is γ, with the same argument n as the concrete side (*i.e.*, $x = n$). The abstract operations of other threads can be known from the precondition p.

For each step of the concrete code C, we require it to be safe, and correspond to some steps of Δ, as shown in the first condition in Definition 5. We define the transition $\Delta \rightrightarrows \Delta'$ as follows.

$$\begin{array}{l} \Delta \rightrightarrows \Delta' \ \text{ iff } \forall U', \theta'.\, (U', \theta') \in \Delta' \\ \qquad\qquad \Longrightarrow\ \exists U, \theta.\, (U, \theta) \in \Delta \wedge (U, \theta) \dashrightarrow^* (U', \theta'), \\ \text{where } (U, \theta) \dashrightarrow (U', \theta') \overset{\text{def}}{=} \exists \mathsf{t}.\, (U, \theta) \dashrightarrow_{\mathsf{t}} (U', \theta') \\ \quad \text{and } (U, \theta) \dashrightarrow_{\mathsf{t}} (U', \theta') \text{ has been defined in Fig. 11.} \end{array}$$

It says, any (U', θ') pair in Δ' should be "reachable" from Δ. Specifically, we could execute the abstract operation of some thread t' (which could be the current thread t or some others), or drop some (U, θ) pair in Δ. The former is like a step of $\textbf{trylin}(\mathsf{t}')$ or $\textbf{lin}(\mathsf{t}')$, depending on whether or not we keep the original abstract operation of t'. The latter can be viewed as a **commit** step, in which we discard the wrong speculations.

We also require the related steps at the two levels to satisfy the guarantee $G * \mathsf{True}$, G for the shared part and True (arbitrary transitions) for the local part. Symmetrically, the second condition in Definition 5 says, the simulation should be preserved under the environment interference $R * \mathsf{Id}$, R for the shared part and Id (identity transitions) for the local part.

Finally, when the method returns (the last condition in Definition 5), we require the current thread t has finished its abstract operation, and the return values match at the two levels.

Like RGSim, our new simulation is *compositional*, thus can ensure contextual refinement between the implementation and the abstract operation, as shown in the following lemma.

Lemma 6 (Simulation Implies Contextual Refinement). For any Π, Γ and φ, if there exist R, G, p and I such that the following hold for all t,

1. for any f such that $\Pi(f) = (x, C)$, we have $\Pi(f) \preceq^{\mathsf{t}}_{R_{\mathsf{t}};G_{\mathsf{t}};p_{\mathsf{t}}} \Gamma(f)$, and $x \notin \mathit{dom}(I)$;
2. $R_{\mathsf{t}} = \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$, $\ I \triangleright \{R_{\mathsf{t}}, G_{\mathsf{t}}\}$, $\ p_{\mathsf{t}} \Rightarrow I$, and $\mathsf{Sta}(p_{\mathsf{t}}, R_{\mathsf{t}})$;
3. $\lfloor \varphi \rfloor \Rightarrow \bigwedge_{\mathsf{t}} p_{\mathsf{t}}$;

then $\Pi \sqsubseteq_\varphi \Gamma$.

Here $x \notin \mathit{dom}(I)$ means the formal argument x is always in the local state, and $\lfloor \varphi \rfloor$ lifts φ to a state assertion:

$$\lfloor \varphi \rfloor \overset{\text{def}}{=} \{(\sigma, \{(\emptyset, \theta)\}) \mid \varphi(\sigma) = \theta\}.$$

Lemma 6 allows us to prove contextual refinement $\Pi \sqsubseteq_\varphi \Gamma$ by showing the simulation $\Pi(f) \preceq^{\mathsf{t}}_{R_{\mathsf{t}};G_{\mathsf{t}};p_{\mathsf{t}}} \Gamma(f)$ for each method f, where R, G and p are defined over the shared states fenced by the invariant I, and the interference constraint $R_{\mathsf{t}} = \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$ holds following Rely-Guarantee reasoning [17]. Its proof is similar to the compositionality proofs of RGSim [19], but now we need to be careful with the helping between threads and the speculations. We give the proofs in TR [18].

Objects	**Helping**	**Fut. LP**	**Java Pkg**	**HS Book**
Treiber stack [29]				√
HSY stack [14]	√			√
MS two-lock queue [23]				√
MS lock-free queue [23]		√	√	√
DGLM queue [6]		√		
Lock-coupling list				√
Optimistic list [15]				√
Heller *et al.* lazy list [13]	√	√		√
Harris-Michael lock-free list	√	√	√	√
Pair snapshot [27]		√		
CCAS [31]	√	√		
RDCSS [12]	√	√		

Table 1. Verified Algorithms Using Our Logic

Lemma 7 (Logic Ensures Simulation for Method). For any t, x, C, γ, R, G and p, if there exist I and $\widetilde{C}$ such that

$$R, G, I \vdash_{\mathsf{t}} \{\mathsf{t} \rightarrowtail (\gamma, x) * p\}\ \widetilde{C}\ \{\mathsf{t} \rightarrowtail (\textbf{end}, \_) * (x = \_) * p\},$$

and $\mathsf{Er}(\widetilde{C}) = (C; \textbf{noret})$, then $(x, C) \preceq^{\mathsf{t}}_{R;G;p} \gamma$.

Here we use $\mathsf{Er}(\widetilde{C})$ to erase the instrumented commands in $\widetilde{C}$. The lemma shows that, verifying $\widetilde{C}$ in our logic establishes simulation between the original code and the abstract operation. It is proved by first showing that our logic ensures the standard rely-guarantee-style partial correctness (see Sec. 4.4). Then we build the simulation by projecting the instrumented semantics (Fig. 11) to the concrete semantics of C (Fig. 5) and the speculative steps $\rightrightarrows$ of Δ.

Finally, from Lemmas 6 and 7, we get the soundness theorem of our logic, which says our logic can verify linearizability.

Theorem 8 (Logic Soundness). For any Π, Γ and φ, if there exist R, G, p and I such that the following hold for all t,

1. for any f, if $\Pi(f) = (x, C)$, there exists $\widetilde{C}$ such that
$$R_{\mathsf{t}}, G_{\mathsf{t}}, I \vdash_{\mathsf{t}} \{\mathsf{t} \rightarrowtail (\Gamma(f), x) * p_{\mathsf{t}}\}\ \widetilde{C}\ \{\mathsf{t} \rightarrowtail (\textbf{end}, \_) * (x = \_) * p_{\mathsf{t}}\},$$
$\mathsf{Er}(\widetilde{C}) = (C; \textbf{noret})$, and $x \notin \mathit{dom}(I)$;
2. $R_{\mathsf{t}} = \bigvee_{\mathsf{t}' \neq \mathsf{t}} G_{\mathsf{t}'}$, $\ p_{\mathsf{t}} \Rightarrow I$, and $\mathsf{Sta}(p_{\mathsf{t}}, R_{\mathsf{t}})$;
3. $\lfloor \varphi \rfloor \Rightarrow \bigwedge_{\mathsf{t}} p_{\mathsf{t}}$;

then $\Pi \sqsubseteq_\varphi \Gamma$, and thus $\Pi \preceq_\varphi \Gamma$.

6. Examples

Our logic gives us an effective approach to verify linearizability. As shown in Table 1, we have verified 12 algorithms, including two stacks, three queues, four lists and three algorithms on atomic memory reads or writes. Table 1 summarizes their features, including the helping mechanism (**Helping**) and future-dependent LPs (**Fut. LP**). Some of them are used in the `java.util.concurrent` package (**Java Pkg**). The last column (**HS Book**) shows whether it occurs in Herlihy and Shavit's classic textbook on concurrent algorithms [15]. We have almost covered all the fine-grained stacks, queues and lists in the book. We can see that our logic supports various objects ranging from simple ones with static LPs to sophisticated ones with non-fixed LPs. Although many of the examples can be verified using other approaches, we provide the first program logic which is proved sound and useful enough to verify all of these algorithms. Their complete proofs are given in TR [18].

In general we verify linearizability in the following steps. First we instrument the code with the auxiliary commands such as **linself**, $\textbf{trylin}(E)$ and $\textbf{commit}(p)$ at proper program points. The instrumentation should not be difficult based on the intuition of the algorithm. Then, we specify the assertions (as in Theorem 8)

```
readPair(int i, j) {  local a, b, v, w;
  $\{I * (\mathtt{cid} \rightarrowtail (\gamma, (\mathtt{i}, \mathtt{j})))\}$
1 while(true) {
    $\{I * (\mathtt{cid} \rightarrowtail (\gamma, (\mathtt{i}, \mathtt{j})) \oplus \mathsf{true})\}$
2   < a := m[i].d; v := m[i].v; >
    $\{\exists v'. (I \wedge \mathsf{readCell}(\mathtt{i}, \mathtt{a}, \mathtt{v}; v')) * (\mathtt{cid} \rightarrowtail (\gamma, (\mathtt{i}, \mathtt{j})) \oplus \mathsf{true})\}$
3   < b := m[j].d; w := m[j].v; trylinself; >
    $\{\exists v'. (I \wedge \mathsf{readCell}(\mathtt{i}, \mathtt{a}, \mathtt{v}; v') \wedge \mathsf{readCell}(\mathtt{j}, \mathtt{b}, \mathtt{w}; \_)) * \mathsf{afterTry}\}$
4   if (v = m[i].v) {
      $\{I * (\mathtt{cid} \rightarrowtail (\mathbf{end}, (\mathtt{a}, \mathtt{b})) \oplus \mathsf{true})\}$
5     commit(cid $\rightarrowtail$ (end, (a, b)));
      $\{I * (\mathtt{cid} \rightarrowtail (\mathbf{end}, (\mathtt{a}, \mathtt{b})))\}$
6     return (a, b);
      $\{I * (\mathtt{cid} \rightarrowtail (\mathbf{end}, (\mathtt{a}, \mathtt{b})))\}$
7   } } }
```

Auxiliary definitions:

$$\mathsf{readCell}(i, d, v; v') \stackrel{\text{def}}{=} (\mathsf{cell}(i, d, v) \vee (\mathsf{cell}(i, \_, v') \wedge v \neq v')) * \mathsf{true}$$

$$\mathsf{absRes} \stackrel{\text{def}}{=} (\mathtt{cid} \rightarrowtail (\mathbf{end}, (\mathtt{a}, \mathtt{b})) \wedge v' = \mathtt{v}) \vee (\mathtt{cid} \rightarrowtail (\mathbf{end}, (\_, \mathtt{b})) \wedge v' \neq \mathtt{v})$$

$$\mathsf{afterTry} \stackrel{\text{def}}{=} \mathtt{cid} \rightarrowtail (\gamma, (\mathtt{i}, \mathtt{j})) \oplus \mathsf{absRes} \oplus \mathsf{true}$$

Figure 12. Proof Outline of `readPair` in Pair Snapshot

and reason about the instrumented code by applying our inference rules, just like the usual partial correctness verification in LRG. In our experience, handling the auxiliary commands usually would not introduce much difficulty over the plain verification with LRG. Below we sketch the proofs of three representative examples: the pair snapshot, MS lock-free queue and the CCAS algorithm.

6.1 Pair Snapshot

As discussed in Sec. 2.3, the pair snapshot algorithm has a future-dependent LP. In Fig. 12, we show the proof of `readPair` for the current thread `cid`. We will use γ for its abstract operation, which atomically reads the cells `i` and `j` at the abstract level.

First, we insert **trylinself** and **commit** as highlighted in Fig. 12. The **commit** command says, when the validation at line 4 succeeds, we must have $\mathtt{cid} \rightarrowtail (\mathbf{end}, (\mathtt{a}, \mathtt{b}))$ as a possible speculation. This actually requires a correct instrumentation of **trylinself**. In Fig. 12, we insert it at line 3. It cannot be moved to other program points since line 3 is the only place where we could get the abstract return value $(\mathtt{a}, \mathtt{b})$ when executing γ. Besides, we cannot replace it by a **linself**, because if line 4 fails later, we have to restart to do the original abstract operation.

After the instrumentation, we can define the precise invariant I, the rely R and the guarantee G. The invariant I simply maps every memory cell (d, v) at the concrete level to a cell with data d at the abstract level, as shown below:

$$I \stackrel{\text{def}}{=} \circledast_{i \in [1..\mathtt{size}]}.(\exists d, v.\, \mathsf{cell}(i, d, v))$$
$$\text{where } \mathsf{cell}(i, d, v) \stackrel{\text{def}}{=} (\mathtt{m}[i] \mapsto (d, v)) * (\mathtt{m}[i] \Mapsto d))$$

Every thread guarantees that when writing a cell, it increases the version number. Here we use $[G]_I$ short for $(G \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$.

$$G \stackrel{\text{def}}{=} [\mathsf{Write}]_I \qquad \mathsf{Write} \stackrel{\text{def}}{=} \exists i, v.\, \mathsf{cell}(i, \_, v) \ltimes \mathsf{cell}(i, \_, v+1)$$

The rely R is the same as the guarantee G.

Then we specify the pre- and post-conditions, and reason about the instrumented code using our inference rules. The proof follows the intuition of the algorithm. Note that we relax $\mathtt{cid} \rightarrowtail (\gamma, (\mathtt{i}, \mathtt{j}))$ in the precondition of the method to $\mathtt{cid} \rightarrowtail (\gamma, (\mathtt{i}, \mathtt{j})) \oplus \mathsf{true}$ to ensure the loop invariant. The latter says, `cid` *may* just start (or restart) its operation and have not done yet.

The `readPair` method in the pair snapshot algorithm is "read-only" in the sense that the abstract operation does not update the abstract object. This perhaps means that it does not matter to linearize the method multiple times. In Sec. 6.3 we will verify an algorithm with future-dependent LPs, CCAS, which is not "read-only". We *can* still "linearize" a method with side effects multiple times.

```
 1 enq(v) {                      16 deq() {
 2  local x, t, s, b;            17  local h, t, s, v, b;
 3  x := cons(v, null);          18  while (true) {
 4  while (true) {               19   h := Head; t := Tail;
 5   t := Tail; s := t.next;     20   s := h.next;
 6   if (t = Tail) {             21   if (h = Head)
 7    if (s = null) {            22    if (h = t) {
 8      b:=cas(&(t.next),s,x);   23      if (s = null)
 9      if (b) {                 24        return EMPTY;
10        cas(&Tail, t, x);      25      cas(&Tail, t, s);
11        return; }              26    }else {
12    }else cas(&Tail, t, s);    27      v := s.val;
13   }                           28      b:=cas(&Head,h,s);
14  }                            29      if(b) return v; }
15 }                             30  } }
```

Figure 13. MS Lock-Free Queue Code

6.2 MS Lock-Free Queue

The widely-used MS lock-free queue [23] also has future-dependent LPs. We show its code in Fig. 13.

The queue is implemented as a linked list with `Head` and `Tail` pointers. `Head` always points to the first node (a sentinel) in the list, and `Tail` points to either the last or second to last node. The `enq` method appends a new node at the tail of the list and advances `Tail`, and `deq` replaces the sentinel node by its next node and returns the value in the new sentinel. If the list contains only the sentinel node, meaning the queue is empty, then `deq` returns `EMPTY`.

The algorithm employs the helping mechanism for the `enq` method to swing the `Tail` pointer when it lags behind the end of the list. A thread should first try to help the half-finished `enq` by advancing `Tail` (lines `12` and `25` in Fig. 13) before doing its own operation. But this helping mechanism would not affect the LP of `enq` which is statically located at line `8` when the `cas` succeeds, since the new node already becomes visible in the queue after being appended to the list, and updating `Tail` will not affect the abstract queue. We simply instrument line `8` as follows to verify `enq`:

```
< b := cas(&(t.next), s, x);  if (b) linself; >
```

On the other hand, the original queue algorithm [23] checks `Head` or `Tail` (line `6` or `21` in Fig. 13) to make sure that its value has not been changed since its local copy was read (at line `5` or `19`), and if it fails, the operation will restart. This check can improve efficiency of the algorithm, but it makes the LP of the `deq` method for the empty queue case depend on future executions. That LP should be at line `20`, if the method returns `EMPTY` at the end of the same iteration. The intuition is, when we read `null` from `h.next` at line `20` (indicating the abstract queue must be empty there), we do not know how the iteration would terminate at that time. If the later check over `Head` at line `21` fails, the operation would restart and line `20` may not be the LP. We can use our try-commit instrumentation to handle this future-dependent LP. We insert **trylinself** at line `20`, as follows:

```
< s := h.next;  if (h = t && s = null) trylinself; >
```

Before the method returns `EMPTY`, we commit to the finished abstract operation, *i.e.*, we insert `commit(cid` $\rightarrowtail$ `(end, EMPTY))` just before line `24`. Also, when we know we have to do another iteration, we can commit to the original DEQ operation, *i.e.*, we insert `commit(cid` $\rightarrowtail$ `DEQ)` at the end of the loop body.

For the case of nonempty queues, the LP of the `deq` method is statically at line `28` when the `cas` succeeds. Thus we can instrument **linself** there, as shown below.

```
< b := cas(&Head, h, s);  if (b) linself; >
```

After the instrumentation, we can define I, R and G and verify the code using our logic rules. The invariant I relates all the nodes

```
 1 CCAS(o, n) {                 11 Complete(d) {
 2  local r, d;                 12  local b;
 3  d := cons(cid, o, n);       13  b := flag;
 4  r := cas(&a, o, d);         14  if (b)
 5  while(IsDesc(r)) {          15    cas(&a, d, d.n);
 6    Complete(r);              16  else
 7    r := cas(&a, o, d);       17    cas(&a, d, d.o);
 8  }                           18 }
 9  if(r = o) Complete(d);      19 SetFlag(b){ flag := b;}
10  return r; }
```

Figure 14. CCAS Code

in the concrete linked list to the abstract queue. R and G specify the related transitions at both levels, which simply include all the actions over the shared states in the algorithm. The proof is similar to the partial correctness proof using LRG, except that we have to specify the abstract objects and operations in assertions and reason about the instrumented code. We show the full proof in TR [18].

6.3 Conditional CAS

Conditional compare-and-swap (CCAS) [31] is a simplified version of the RDCSS algorithm [12]. It involves both the helping mechanism and future-dependent LPs. We show its code in Fig. 14.

The object contains an integer variable `a`, and a boolean bit `flag`. The method `SetFlag` (line 19) sets the bit directly. The method `CCAS` takes two arguments: an expected current value `o` of the variable `a` and a new value `n`. It atomically updates `a` with the new value if `flag` is true and `a` indeed has the value `o`; and does nothing otherwise. `CCAS` always returns the old value of `a`.

The implementation in Fig. 14 uses a variant of `cas`: instead of a boolean value indicating whether it succeeds, `cas(&a,o,n)` returns the old value stored in `a`. When starting a `CCAS`, a thread first allocates its descriptor (line 3), which contains the thread id and the arguments for CCAS. It then tries to put its descriptor in `a` (line 4). If successful (line 9), it calls the auxiliary `Complete` function, which restores `a` to the new value `n` (line 15) or to the original value `o` (line 17), depending on whether `flag` is true. If it finds `a` contains a descriptor (*i.e.*, `IsDesc` holds), it will try to help complete the operation in the descriptor (line 6) before doing its own. Since we disallow nested function calls to simplify the language, the auxiliary `Complete` function should be viewed as a macro.

The LPs of the algorithm are at lines 4, 7 and 13. If `a` contains a different value from `o` at lines 4 and 7, then CCAS fails and they are LPs of the current thread. We can instrument these lines as follows:

```
<r := cas(&a, o, d);  if(r!=o && !IsDesc(r)) linself;>
```

If the descriptor `d` gets placed in `a`, then the LP should be in the `Complete` function. Since any thread can call `Complete` to help the operation, the LP should be at line 13 of the thread which will succeed at line 15 or 17. It is a future-dependent LP which may be in other threads' code. We instrument line 13 using **trylin**(`d.id`) to speculatively execute the abstract operation for the thread in `d`, which may not be the current thread. That is, line 13 becomes:

```
< b := flag;  if (a = d) trylin(d.id); >
```

The condition `a=d` requires that the abstract operation in the descriptor has not been finished. Then at lines 15 and 17, we commit the correct guess. We show the instrumentation at line 15 below (where `s` is a local variable), and line 17 is instrumented similarly.

```
< s := cas(&a, d, d.n);
  if(s = d) commit(d.id ↣(end, d.o) * a⤇d.n); >
```

That is, it should be possible that the thread in `d` has finished the operation, and the current abstract `a` is the new value `n`.

Then we can define I, R and G, and verify the code by applying the inference rules. The invariant I says, the shared state includes `flag` and `a` at the abstract and the concrete levels; and when `a` is a descriptor `d`, the descriptor and the abstract operation of the thread `d.id` are also shared.

The rely R and the guarantee G should include the action over the shared states at each line. The action at line 4 (or 7) is interesting. If it succeeds, *both* the descriptor `d` and the abstract operation will be transferred from the local state to the shared part. This puts the abstract operation in the pending thread pool and enables other threads to help execute it.

The action at line 13 guarantees $\mathsf{TrylinSucc} \vee \mathsf{TrylinFail}$, which demonstrates the use of our logic for both helping and speculation.

$$\begin{aligned}\mathsf{TrylinSucc} &\overset{\text{def}}{=} (\exists t, o, n.\, (\texttt{flag} \Mapsto \textbf{true} * \mathsf{notDone}(t,o,n)) \\ &\qquad \ltimes (\texttt{flag} \Mapsto \textbf{true} * \mathsf{endSucc}(t,o,n))) \oplus \mathsf{Id} \\ \text{where } \mathsf{notDone}(t,o,n) &\overset{\text{def}}{=} t \rightarrowtail (\mathrm{CCAS}, o, n) * \mathsf{a} \Mapsto o \\ \mathsf{endSucc}(t,o,n) &\overset{\text{def}}{=} t \rightarrowtail (\textbf{end}, o) * \mathsf{a} \Mapsto n\end{aligned}$$

$\mathsf{TrylinFail}$ is symmetric for the case when `flag` $\Mapsto$ **false**. Here we use $R \oplus \mathsf{Id}$ (defined in Fig. 8) to describe the action of **trylin**. It means, after the action we will keep the original state as well as the new state resulting from R as possible speculations. Also, in $\mathsf{TrylinSucc}$ and $\mathsf{TrylinFail}$, the current thread is allowed to help execute the abstract CCAS of some thread t.

The subtle part in the proof is to ensure that, no thread could cheat by imagining another thread's help. In any program point of CCAS, the environment may have done **trylin** and helped the operation. But whether the environment has helped it or not, the **commit** at line 15 or 17 cannot fail. This means, we should not confuse the two kinds of nondeterminism caused by speculation and by environment interference. The former allows us to discard wrong guesses, while for the latter, we should consider *all* possible environments (including none).

7. Related Work and Conclusion

In addition to the work mentioned in Sec. 1 and 2, there is a large body of work on linearizability verification. Here we only discuss the most closely related work that can handle non-fixed LPs.

Our logic is similar to Vafeiadis' extension of RGSep to prove linearizability [32]. He also uses abstract objects and abstract atomic operations as auxiliary variables and code. There are two key differences between the logics. First he uses prophecy variables to handle future-dependent LPs, but there has been no satisfactory semantics given for prophecy variables so far. We use the simple try-commit mechanism, whose semantics is straightforward. Second the soundness of his logic *w.r.t.* linearizability is not specified and proved. We address this problem by defining a new thread-local simulation as the meta-theory of our logic. As we explained in Sec. 2, defining such a simulation to support non-fixed LPs is one of the most challenging issues we have to solve. Although recently Vafeiadis develops an automatic verification tool [33] with formal soundness for linearizability, his new work can handle non-fixed LPs for *read-only* methods only, and cannot verify algorithms like HSY stack, CCAS and RDCSS in our paper.

Recently, Turon *et al.* [31] propose logical relations to verify fine-grained concurrency, which establish contextual refinement between the library and the specification. Underlying the model a similar simulation is defined. Our pending thread pool is proposed concurrently with their "spec thread pool", while the speculation idea in our simulation is borrowed from their work, which traces back to forward-backward simulation [21]. What is new here is a new program logic and the way we instrument code to do relational reasoning. The set of syntactic rules, including the try-commit mechanism to handle uncertainty, is much easier to use than the semantic logical relations to construct proofs. On the other hand, they support higher-order features, recursive types and polymorphism, while we focus on concurrency reasoning and use only a simple first-order language.

O'Hearn *et al.* [25] prove linearizability of an optimistic variant of the lazy set algorithm by identifying the "Hindsight" property of the algorithm. Their Hindsight Lemma provides a *non-constructive* evidence for linearizability. Although Hindsight can capture the insights of the set algorithm, it remains an open problem whether the Hindsight-like lemmas exist for other concurrent algorithms.

Colvin *et al.* [3] formally verify the lazy set algorithm using a combination of forward and backward simulations between automata. Their simulations are not thread-local, where they need to know the program counters of all threads. Besides, their simulations are specifically constructed for the lazy set only, while ours is more general in that it can be satisfied by various algorithms.

The simulations defined by Derrick *et al.* [4] are thread-local and general, but they require the operations with non-fixed LPs to be read-only, thus cannot handle the CCAS example. They also propose a backward simulation to verify linearizability [28]. Although the method is proved to be complete, it does not support thread-local verification and there is no program logic given.

Elmas *et al.* [7] prove linearizability by incrementally *rewriting* the fine-grained code to the atomic operation. They do not need to locate LPs. Their rules are based on left/right movers and program refinements, but not for Hoare-style reasoning as in our work.

There are also lots of model-checking based tools (*e.g.*, [20, 34]) for *checking* linearizability. For example, Vechev *et al.* [34] check linearizability with user-specified non-fixed LPs. Their method is not thread modular. To handle non-fixed LPs, they need users to instrument the code with enough information about the actions of other threads, which usually demands a priori knowledge about the number of threads running in parallel, as shown in their example. Besides, although their checker can detect un-linearizable code, it will not terminate for linearizable methods in general.

Conclusion. We propose a new program logic to verify linearizability of algorithms with non-fixed LPs. The logic extends LRG [8] with new rules for the auxiliary commands introduced specifically for linearizability proof. We also give a relational interpretation of asssertions and rely/guarantee conditions to relate concrete implementations with the corresponding abstract operations. Underlying the logic is a new thread-local simulation, which gives us contextual refinement. Linearizability is derived based on its equivalence to refinement. Both the logic and the simulation support reasoning about the helping mechanism and future-dependent LPs. As shown in Table 1, we have applied the logic to verify various classic algorithms.

Acknowledgments

We would like to thank Matthew Parkinson, Zhong Shao, Jan Hoffmann 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 No. 61073040 and 61229201, the National Hi-Tech Research and Development Program of China (Grant No. 2012AA010901), and Program for New Century Excellent Talents in Universities (Grant No. NCET-2010-0984). Part of the work is done during Hongjin Liang's visit to Yale University in 2012-2013, which is supported by China Scholarship Council.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[2] D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV'07*.

[3] R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set algorithm. In *CAV'06*.

[4] J. Derrick, G. Schellhorn, and H. Wehrheim. Verifying linearisability with potential linearisation points. In *FM'11*.

[5] J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *ACM TOPLAS*, 33(1):4, 2011.

[6] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE'04*.

[7] T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Tasiran. Simplifying linearizability proofs with reduction and abstraction. In *TACAS'10*.

[8] X. Feng. Local rely-guarantee reasoning. In *POPL'09*.

[9] I. Filipović, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 2010.

[10] A. Gotsman and H. Yang. Linearizability with ownership transfer. In *CONCUR'12*.

[11] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC'01*.

[12] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *DISC'02*.

[13] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. S. III, and N. Shavit. A lazy concurrent list-based set algorithm. In *OPODIS'05*.

[14] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA'04*.

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

[16] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.

[17] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983.

[18] H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. Technical report, USTC, March 2013. `http://kyhcs.ustcsz.edu.cn/relconcur/lin`.

[19] H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL'12*.

[20] Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model checking linearizability via refinement. In *FM'09*.

[21] N. A. Lynch and F. W. Vaandrager. Forward and backward simulations: I. untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.

[22] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA'02*.

[23] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC'96*.

[24] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[25] P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight. In *PODC'10*, .

[26] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL'04*, .

[27] S. Qadeer, A. Sezgin, and S. Tasiran. Back and forth: Prophecy variables for static verification of concurrent programs. Tech Report.

[28] G. Schellhorn, H. Wehrheim, and J. Derrick. How to prove algorithms linearisable. In *CAV'12*.

[29] R. K. Treiber. System programming: coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

[30] A. Turon and M. Wand. A separation logic for refining concurrent objects. In *POPL'11*.

[31] A. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In *POPL'13*.

[32] V. Vafeiadis. Modular fine-grained concurrency verification. Thesis.

[33] V. Vafeiadis. Automatically proving linearizability. In *CAV*, 2010.

[34] M. T. Vechev, E. Yahav, and G. Yorsh. Experience with model checking linearizability. In *SPIN'09*.