Mover Logic: A Concurrent Program Logic for Reduction and Rely-Guarantee Reasoning

Cormac Flanagan
University of California, Santa Cruz, CA, USA

Stephen N. Freund
Williams College, Williamstown, MA, USA

Abstract

Rely-guarantee (RG) logic uses thread interference specifications (relies and guarantees) to reason about the correctness of multithreaded software. Unfortunately, RG logic requires each function postcondition to be "stabilized" or specialized to the behavior of other threads, making it difficult to write function specifications that are reusable at multiple call sites.

This paper presents mover logic, which extends RG logic to address this problem via the notion of atomic functions. Atomic functions behave as if they execute serially without interference from concurrent threads, and so they can be assigned more general and reusable specifications that avoid the stabilization requirement of RG logic. Several practical verifiers (Calvin-R, QED, CIVL, Armada, Anchor, etc.) have demonstrated the modularity benefits of atomic function specifications. However, the complexity of these systems and their correctness proofs makes it challenging to understand and extend these systems. Mover logic formalizes the central ideas of reduction in a declarative program logic that provides a foundation for future work in this area.

2012 ACM Subject Classification Theory of computation → Program verification; Theory of computation → Program specifications; Software and its engineering → Concurrent programming languages; Software and its engineering → Formal software verification

Keywords and phrases concurrent program verification, reduction, rely-guarantee reasoning, synchronization

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.16

Related Version *Extended Version*: `https://arxiv.org/abs/2407.08070` [20]

Funding This work was supported by NSF grants 2243636 and 2243637.

1 Introduction

Verifying that a multithreaded software system behaves correctly for all possible inputs and thread interleavings is a critically important problem in computer science. To verify large systems, verification techniques must employ *modular reasoning* in which each function's implementation is verified with respect to its specification. In a multithreaded system, writing precise and reusable function specifications is a rather difficult challenge, since concurrent threads can observe and change the state of a function call not just in its initial and final states, but also at any intermediate states during the function's execution. Thus, function specifications must describe not just the function's precondition and postcondition, but also how the function may influence and be influenced by other concurrent threads. To address this problem, Rely-Guarantee (RG) logic [33] uses function specifications that include:

- a *guarantee* G describing how each step of the function may update shared state, and
- a *rely assumption* R describing the behavior of interleaved steps of other threads. The rely assumption might, for example, specify that interleaved steps preserve a data invariant.

Under RG logic, however, a function's postcondition must summarize not only the behavior the function itself but also the behavior of interleaved steps of other threads [56]. Consequently, RG function specifications are often specialized to the rely assumption and data invariants of a particular client, limiting reuse of those function specifications in other clients, as we illustrate in Section 2.

Lipton's theory of reduction [41] provides a promising approach to address this problem. It uses a commuting argument to show that certain functions are *atomic* and behave as if they execute serially (without interleaved steps of other threads). Consequently, atomic functions do not require interleaved rely assumptions, and they can be precisely specified using preconditions and postconditions that are independent of any specific client.

Reduction has been widely adopted in a variety of software validation tools, including dynamic analyses [17, 54, 55, 9], type systems [50, 24, 23, 22], and other tools [6, 61, 62, 60]. Over the past two decades, software verifiers based on reduction (*e.g.*, Calvin-R [25], QED [15], CIVL [30, 38], Armada [42], and Anchor [19]) have demonstrated the utility of atomic function specifications in verifying sophisticated concurrent code. To date, however, reduction-based verifiers have not been based on an underlying program logic, such as RG logic. Instead, their soundness arguments are typically based on monolithic proofs whose complexity inhibits further research. To address this complexity barrier, we present mover logic, which extends RG logic to support atomic function specifications via reduction-based reasoning.

In mover logic, thread interference points are documented with `yield` annotations that have no run-time effect. Mover logic verifies that every sequence of operations between two yield points is reducible and hence amenable to sequential reasoning. In order to verify reducibility, mover logic uses synchronization specifications describing both when each thread can access each shared location and how those accesses commute with concurrent accesses of other threads. In contrast to RG logics that must stabilize all state predicates under the rely assumption, mover logic only needs to stabilize predicates at `yield` points. Atomic functions have no yield points and can thus be specified with traditional pre- and postconditions. Moreover, atomic function specifications need not include a client-specific rely assumption that would limit reuse in other clients that have different rely assumptions or data invariants.

Mover logic is a declarative program logic (similar in style to Hoare Logic and RG Logic) that helps explain and justify many subtle aspects of reduction-based verification, including:

- what code blocks are reducible;
- where yield annotations are required;
- which functions are atomic;
- what atomic and non-atomic function specifications mean;
- what reasoning is performed by the verifier;
- why this reasoning is sound; and
- which programs are verifiable or not verifiable, and why.

Mover logic simplifies the soundness proof for any specific verifier, because the proof now must only show that the verifier follows the rules of mover logic.

The presentation of our results proceeds as follows.

- Section 2 illustrates the specification entanglement problem of RG logic and shows how mover logic avoids this problem.
- Section 3 reviews Lipton's theory of reduction.
- Sections 4 and 5 present an overview of mover logic and additional examples.
- Section 6 formalizes a core multithreaded language.
- Section 7 and 8 present mover logic for this language.
- Sections 9 and 10 discuss related work and summarize our contributions.

For clarity of exposition, our presentation of mover logic targets an idealized multithreaded language that captures the essential complexities of multithreaded function specifications. Extending the logic to more complex languages remains an important topic for future work.

2 Limitations of Rely-Guarantee Logic

We motivate the need for mover logic via the example code in Figure 1 (left). That code consists of:

1. A **counter library** that contains the function `add(n)` that adds `n` to the variable `x` and returns the new value of `x`. The initial value of variable `x` is 0, and it is protected by lock `m`, whose value is either the thread identifier *tid* of the thread holding the lock or 0 if unheld. The lock is initially unheld.
2. A **first client** that creates two threads, and each thread calls `add(2)` multiple times before asserting that `x` is even.

This program verifies under RG logic based on the invariant that `x` is always even. This `even(x)` invariant is a precondition and postcondition for both `add()` and `client()`[1]:

```
requires even(x)
ensures  even(x)
```

In addition, each step by each thread in the program is guaranteed to preserve this invariant. As a result, each thread can rely on other threads to preserve the invariant:

```
relies     even(x)
guarantees even(x)
```

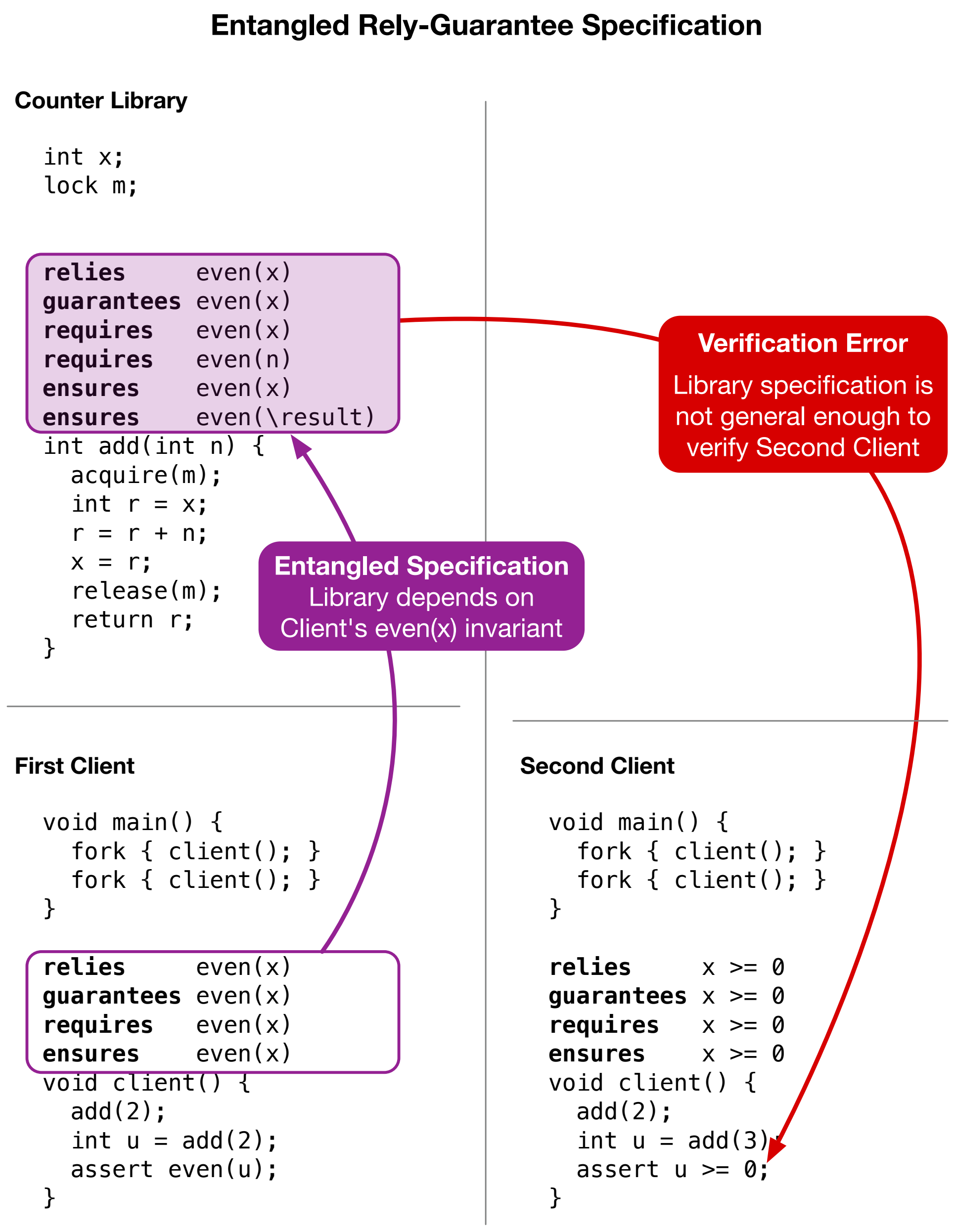
These RG specifications are sufficient to verify that the program does not go wrong by failing the `even(u)` assertion in `client()`, but unfortunately the specification for `add()` is tightly-coupled, or *entangled*, with the `even(x)` data invariant from this particular client. A different client would necessitate revising and re-verifying the counter library, which makes modular verification more challenging and less scalable. For example, the second client in Figure 1 (right) enforces the data invariant `x >= 0`, but it cannot be verified with the `add()` specification entangled with the first client. Others have noted this limitation as well (see, for example, [14, 56]).

2.1 Disentangling RG Specifications: First Attempt

The goal of this paper is to support specifications for library functions like `add()` that are *not* specialized to one particular client. As a first attempt to achieve that goal, the code in Figure 2 (left) uses the following natural postconditions for `add()`, where `\old(x)` and `x` refer to the value of `x` upon function entry and exit, respectively:

```
ensures x == \old(x) + n
ensures \result == x
```

[1] Frame conditions, which specify the locations a function may read or modify, also play a key role in modular function specifications, but we do not consider them in this paper due to lack of space. Extending mover logic with frame conditions, perhaps using ideas from separation logic [47, 49], remains an important topic for future work.


Entangled Rely-Guarantee Specification
Counter Library
First Client
Second Client
Verification Error
Library specification is not general enough to verify Second Client
Entangled Specification
Library depends on Client's even(x) invariant

```
int x;
lock m;

relies     even(x)
guarantees even(x)
requires   even(x)
requires   even(n)
ensures    even(x)
ensures    even(\result)
int add(int n) {
  acquire(m);
  int r = x;
  r = r + n;
  x = r;
  release(m);
  return r;
}
```

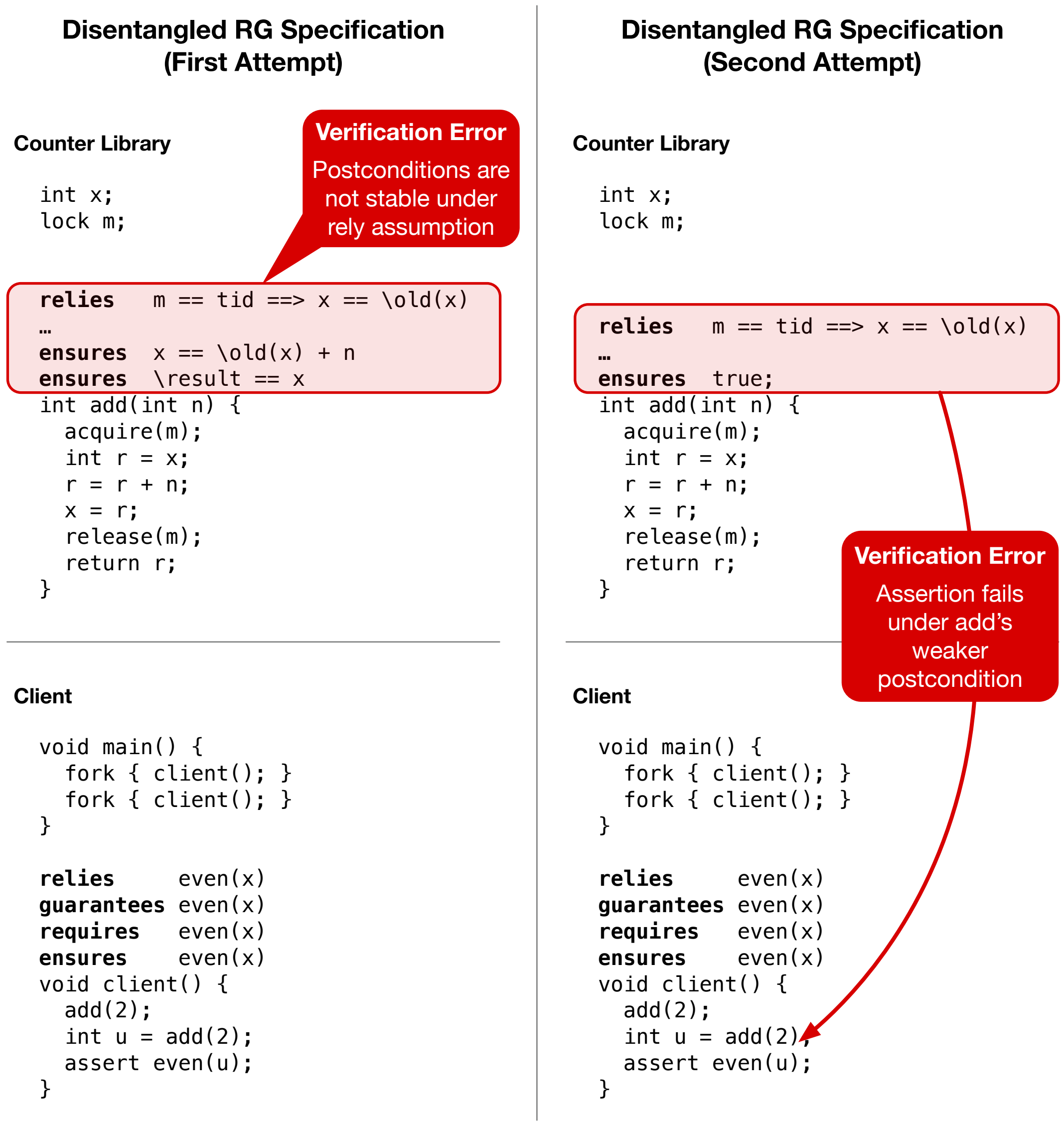
```
void main() {
  fork { client(); }
  fork { client(); }
}

relies     even(x)
guarantees even(x)
requires   even(x)
ensures    even(x)
void client() {
  add(2);
  int u = add(2);
  assert even(u);
}
```

```
void main() {
  fork { client(); }
  fork { client(); }
}

relies     x >= 0
guarantees x >= 0
requires   x >= 0
ensures    x >= 0
void client() {
  add(2);
  int u = add(3);
  assert u >= 0;
}
```

Figure 1 Our idealized running example is an `add(n)` library function that atomically increases shared variable `x` by `n`. **(Left)** A rely-guarantee specification. The client's data invariant `even(x)` becomes entangled in the library specification. **(Right)** A second client that cannot be verified because the specification is insufficiently general.


Disentangled RG Specification (First Attempt)
Verification Error
Postconditions are not stable under rely assumption
Disentangled RG Specification (Second Attempt)
Verification Error
Assertion fails under add's weaker postcondition

Counter Library

```
int x;
lock m;

relies    m == tid ==> x == \old(x)
…
ensures  x == \old(x) + n
ensures  \result == x
int add(int n) {
  acquire(m);
  int r = x;
  r = r + n;
  x = r;
  release(m);
  return r;
}
```

Client

```
void main() {
  fork { client(); }
  fork { client(); }
}

relies     even(x)
guarantees even(x)
requires   even(x)
ensures    even(x)
void client() {
  add(2);
  int u = add(2);
  assert even(u);
}
```

Counter Library

```
int x;
lock m;

relies    m == tid ==> x == \old(x)
…
ensures  true;
int add(int n) {
  acquire(m);
  int r = x;
  r = r + n;
  x = r;
  release(m);
  return r;
}
```

Client

```
void main() {
  fork { client(); }
  fork { client(); }
}

relies     even(x)
guarantees even(x)
requires   even(x)
ensures    even(x)
void client() {
  add(2);
  int u = add(2);
  assert even(u);
}
```

Figure 2 (Left) An attempt to disentangle the library specification from the client that does not meet RG stability requirements. **(Right)** Another attempt that meets stability requirements but fails to verify the client.

In addition, if `add()` has no knowledge of its client, it must assume that other client threads could call `add()` with arbitrary arguments at any time, and so the natural rely assumption is that other threads may update `x` whenever the lock `m` is not held by the current thread. That assumption is most easily expressed as its contra-positive (where `tid` is the identifier of the current thread and lock `m` is held by that thread when `m == tid`):

```
relies  m == tid ==> x == \old(x)
```

Here, `\old(x)` and `x` refer to the value of `x` before and after an interleaved action of another thread, respectively.

To account for interleaved steps of other threads, a central requirement of RG logic is that all store predicates (*e.g.* preconditions, postconditions, and invariants) used to reason about program behavior must be *stable* under this rely assumption R. This means that interleaved R-steps from other threads must not invalidate those predicates. In the case of `add` in Figure 2 (left), the postcondition `x == \old(x) + n && \result == x` is not stable under the rely assumption R, reflecting that `x` could be concurrently modified after the lock is released but before `add()` returns. Thus, Figure 2 (left) does not verify under RG logic.

2.2 Disentangling RG Specifications: Second Attempt

To ensure stability we must weaken the `add()` function's postcondition to be stable under the rely assumption, as shown in Figure 2 (right). Unfortunately, the resulting stable post condition is simply `true`, which no longer guarantees anything about the value of `x` and is too weak to verify the client.

2.3 Broken Invariants and Bidirectional Entanglement

As a more challenging example, consider the `add()` library variant in Figure 3 (left) that temporarily breaks the `even(x)` invariant while holding the lock. In this case, the invariant holds only when lock `m` is free:

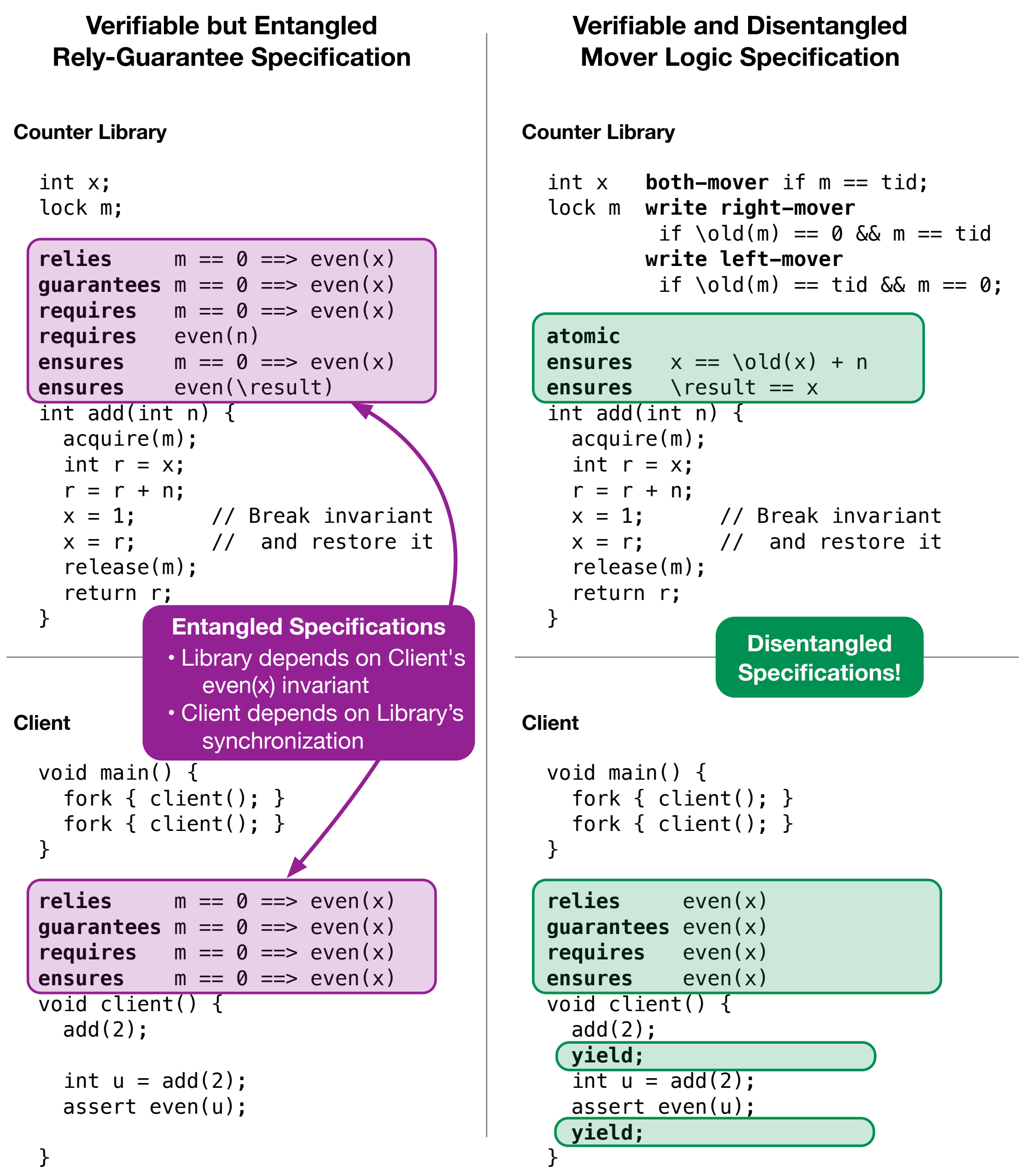
```
m == 0 ==> even(x)
```

Stores at the program points in which the invariant is broken are not intended to be observable by clients. However, the revised RG specifications for `add()` and the client must now be based on this conditional invariant, resulting in two problems. First, the library specification is again specialized to the client's `even(x)` invariant. Second, the library's internal locking discipline leaks into the client's specification, limiting our ability to modify the library code without breaking clients. This example demonstrates that RG reasoning may force us to lose modularity between client and library.

3 Review of Lipton's Theory of Reduction

Our solution to this specification problem employs Lipton's theory of reduction [41], which classifies how steps of one thread commute with concurrent steps of another thread.

- A step is a *right-mover* (`R`) if it commutes "to the right" of any subsequent step by a different thread, in that performing the steps in the opposite order does not change the final store. A lock acquire is a right-mover because any subsequent step from another thread cannot modify that lock.


Verifiable but Entangled
Rely-Guarantee Specification
Verifiable and Disentangled
Mover Logic Specification
Counter Library
Counter Library
Entangled Specifications
• Library depends on Client's even(x) invariant
• Client depends on Library's synchronization
Disentangled Specifications!
Client
Client

```
int x;
lock m;

relies     m == 0 ==> even(x)
guarantees m == 0 ==> even(x)
requires   m == 0 ==> even(x)
requires   even(n)
ensures    m == 0 ==> even(x)
ensures    even(\result)
int add(int n) {
  acquire(m);
  int r = x;
  r = r + n;
  x = 1;       // Break invariant
  x = r;       //   and restore it
  release(m);
  return r;
}
```

```
void main() {
  fork { client(); }
  fork { client(); }
}

relies     m == 0 ==> even(x)
guarantees m == 0 ==> even(x)
requires   m == 0 ==> even(x)
ensures    m == 0 ==> even(x)
void client() {
  add(2);

  int u = add(2);
  assert even(u);

}
```

```
int x   both-mover if m == tid;
lock m  write right-mover
         if \old(m) == 0 && m == tid
        write left-mover
         if \old(m) == tid && m == 0;

atomic
ensures   x == \old(x) + n
ensures   \result == x
int add(int n) {
  acquire(m);
  int r = x;
  r = r + n;
  x = 1;       // Break invariant
  x = r;       //   and restore it
  release(m);
  return r;
}
```

```
void main() {
  fork { client(); }
  fork { client(); }
}

relies     even(x)
guarantees even(x)
requires   even(x)
ensures    even(x)
void client() {
  add(2);
  yield;
  int u = add(2);
  assert even(u);
  yield;
}
```

Figure 3 A second version of the counter library with a temporarily broken `even(x)` invariant. **(Left)** Under RG logic, the library specification is entangled with the client's `even(x)` invariant and the client specification is entangled with the library's synchronization discipline. **(Right)** Under mover logic, the specifications are cleanly disentangled.

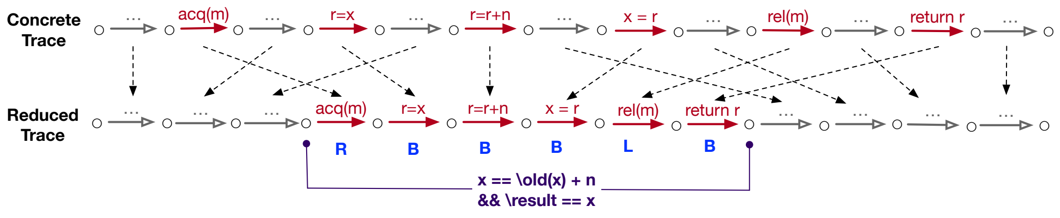


Figure 4 Reduction applied to an execution trace of `add()` from Figure 1.

- Conversely, a step is a *left-mover* (L) if it commutes "to the left" of a preceding step of a different thread. A lock release is a left-mover because any preceding step cannot modify that lock.
- A step is a *both-mover* (B) if it is both a left- and a right-mover, and it is a *non-mover* (N) if neither. A race-free memory access is a both-mover because there are no concurrent, conflicting accesses. An access to a race-prone variable is a non-mover since there may be concurrent writes.

A sequence of steps performed by a particular thread is *reducible* if consists of (1) zero or more right-movers; (2) at most one non-mover; and (3) zero or more left-movers. That is, a sequence is reducible if the commutativity of the steps match the pattern R*[N]L*. Any interleaved steps of other threads can be "commuted out" to produce a serial execution.

Figure 4 illustrates this technique for a call to `add()` interleaved with steps of a second thread. In that figure and below, the solid and hollow arrow heads indicate steps from different threads, and arrows labeled "..." represent any number of steps by that thread. The steps of `add()` have the mover behavior R B B B L B, matching the reducible pattern R*[N]L*. Thus we can reason about `add()` as if it executes sequentially and assign it the intuitive postcondition `x == \old(x) + n && \result == x`.

4 Overview of Mover Logic

Mover logic extends RG logic to verify that certain functions are *atomic* and can therefore be assigned more precise (unstabilized) postconditions than under RG logic. Figure 3 (right) shows a mover logic specification for our library/client example. The declaration

```
int x  both-mover if m == tid;
```

means that accesses to `x` are both-movers provided that the current thread holds lock `m`. All other accesses are errors. The declaration for lock `m` specifies that acquires (which change `m` from 0 to the current thread's identifier `tid`) are right-movers and releases (which change `m` from `tid` back to 0) are left-movers:

```
lock m  write right-mover
          if \old(m) == 0 && m == tid
        write left-mover
          if \old(m) == tid && m == 0;
```

These mover specifications are sufficient to verify that `add` is atomic. Consequently, there is no need to apply the rely assumption at each intermediate store inside this atomic function. Instead, sequential reasoning suffices to establish the desired postcondition `x == \old(x) + n && \result == x`.

The `client()` function in Figure 3 (right) is not atomic because steps of other threads could be interleaved between the two calls to `add(2)`. Mover logic uses a `yield` annotation to identify that thread interference may occur at that point, and the store invariants at `yield`s must be stable under the rely assumption:

```
relies      even(x);
guarantees  even(x);
```

Note that this thread guarantee does not need to summarize individual steps inside the callee `add()`, which would expose the broken invariant. Instead, it summarizes the entire atomic effect of `add()`, which preserves the `even(x)` invariant. With mover logic, the `client()` specification is independent of the internal synchronization discipline inside `add()`.

This library/client example illustrates several benefits of mover logic:

- Verifying that `add()` is atomic enables sequential reasoning inside `add()`.
- We thus avoid applying the rely assumption at each program point inside `add()`.
- As a result, `add()` satisfies the desired postcondition `x == \old(x) + n && \result == x`, which is independent of the client-specific data invariant `even(x)`.
- On the client side, the thread guarantee `even(x)` summarizes the entire behavior of `add()`, rather than the behavior of each individual step.
- Consequently, the client can be verified based on the illusion that `even(x)` always holds, with no loss of soundness.

Thus, mover logic disentangles the library specification from the data invariant of the client while also disentangling the client specification from the library synchronization discipline.

5 Additional Examples

5.1 Spin Lock

To further illustrate the benefits of disentangled specifications, Figure 5 (left) shows our counter library rewritten to employ a user-defined spin lock. The `spin_lock()` code employs a compare-and-set operation (`cas`) to attempt to change the lock `l` from 0 to the current thread's `tid`. The `cas` operation returns true if the update succeeds, and false otherwise. Thus, the function retries until the update is success, at which point the current thread holds the lock. The `spin_unlock()` function releases the lock by setting `l` back to 0.

Mover logic verifies that calls to `spin_lock()` and `spin_unlock()` are atomic right- and left-movers, respectively. That enables us to avoid entangled specifications for the spin lock and counter libraries, and the counter library's `add` specification is *identical* to the earlier implementation. It is still atomic and it guarantees the same post condition.

5.2 Lock-Free Queue

Figure 5 (top right) shows a lock-free single-element queue, where `buf` holds either the single enqueued `int` or `None` if the queue is empty, as indicated by the declared type `Optional[int]`.

The `enqueue(v)` function uses `cas` to switch `buf` from `None` to `v` and is atomic since failing `cas` operations are both-movers. The `dequeue()` function use the action `r ~= buf` to denote an *unstable read* of `buf` that can load any value into the local variable `r` [22]. Unstable reads can be treated as right-movers since they commute past steps by other

Verifiable Spin Lock, Counter, Client

Spin Lock Library

```
int l write right-mover
        if \old(l) == 0 && l == tid
      write left-mover
        if \old(l) == tid && l == 0;

atomic right-mover
ensures l == tid
void spin_lock() {
  while (!cas(l, 0, tid)) {
    skip;
  }
}

atomic left-mover
requires l == tid
void spin_unlock() {
  l = 0;
}
```

Disentangled Specifications!

Counter Library

```
int x   both-mover if l == tid;

atomic
ensures x == \old(x) + n
ensures \result == x
int add(int n) {
  spin_lock();
  int r = x;
  r = r + n;
  x = 1;      // Break invariant
  x = r;      //  and restore it
  spin_unlock();
  return r;
}
```

Disentangled Specifications!

Client

```
void main() {
  fork { client(); }
  fork { client(); }
}

relies     even(x)
guarantees even(x)
requires   even(x)
ensures    even(x)
void client() {
  add(2);
  yield;
  int u = add(2);
  assert even(u);
  yield;
}
```

Verifiable Lock-Free Queue Library

```
Optional[int] buf non-mover;

atomic
requires n != None
ensures buf == n
void enqueue(int n) {
  while (!cas(buf, None, n)) {
    skip;
  }
}

atomic
ensures \result == \old(buf)
ensures buf == None
int dequeue() {
  Optional[int] r ~= buf;
  while (r == None) { r ~= buf; }

  while (!cas(buf, r, None)) {
     r ~= buf;
     while (r == None) { r ~= buf; }
  }
  return r;
}
```

Verifiable Lock-Free Stack Library

```
List top  non-mover;

atomic
ensures head(top) == v
ensures tail(top) == \old(top)
void push(int v) {
  List t ~= top;
  List nu = v::t;
  while (!cas(top, t, nu)) {
    t ~= top;
    nu = v::t;
  }
}

atomic
ensures head(\old(top)) == \result
ensures tail(\old(top)) == top
int pop() {
  List t ~= top;
  while (t == Nil) { t ~= top; }
  List tl = tail(t);

  while (!cas(top, t, tl) {
    t ~= top;
    while (t == Nil) { t ~= top; }
    tl = tail(t);
  }
  return head(t);
}
```

Figure 5 (Left) A new implementation of the counter library using a user-defined spin lock. **(Top Right)** A single-element lock-free queue. **(Bottom Right)** A lock-free stack.

threads.[2] Consequently, the `dequeue()` function is atomic. All executions of that function consist of unstable reads (right-movers) and failed `cas` operations (both-movers) followed by a successful `cas` (non-mover). These sequences match the reducible pattern $\texttt{R}^*[\texttt{N}]\texttt{L}^*$. Moreover, the final `cas` ensures that `r` is equal to the pre-`cas` value of `buf`, which enables mover logic to establish the desired post-conditions `\result == \old(buf)` and `buf == None`.

5.3 Lock-Free Stack

Figure 5 (bottom right) shows a lock-free stack. This examples uses immutable lists, where `Nil` is the empty list, v`::`s adds v to the front of the the list s, and `head`(s) and `tail`(s) extract the first element and the rest of s, respectively.[3]

The `push(v)` function is atomic since it has only one non-mover operation, namely the successful `cas`. The unstable reads, list allocations `v::t`, and failed `cas` operations are both-movers or right-movers. Therefore, we can assign `push(v)` the following intuitive post-condition without needing to stabilize under the rely assumption of a particular caller.

```
ensures head(top) == v
ensures tail(top) == \old(top)
```

The `pop()` function is also atomic due to similar reasoning and satisfies the following post-condition without the need to stabilize it.

```
ensures head(\old(top)) == \result
ensures tail(\old(top)) == top
```

6 Mover Logic Language

We formalize mover logic for the idealized language MML (mover logic language), which we summarize in Figure 6. Section 7.3 below translates our running example into MLL. In MLL, threads manipulate a shared store σ that maps variables to values. Variables include x,y,z, and m. We often use the variable m as a lock, where m is the thread identifier (tid) of the thread holding the lock, or 0 if it is not held.

Thread-local variables r are supported by having each thread access a separate variable r_{tid} for each thread tid. The language includes reads and writes to global and local variables, acquires and releases of locks, local computations, etc. For generality and simplicity, we abstract all of these store-manipulation operations as *actions* $A \subseteq \mathit{Tid} \times \mathit{Store} \times \mathit{Store}$. Note that an action may depend on the current thread's identifier. We write actions as formulae in which `\old`(x) and x to refer to the values of x in the pre-store and post-store, respectively. We write $\langle A\rangle_x$ to denote an action that only changes x:

$$\langle A\rangle_x \stackrel{\text{def}}{=} \{\ (\mathit{tid},\sigma,\sigma') \ \mid\ (\mathit{tid},\sigma,\sigma')\in A\ \wedge\ \forall y\in \mathit{Var}.\ y\neq x \Rightarrow \sigma(y)=\sigma'(y)\ \}$$

[2] Unstable reads are a proof technique that trades off our ability to reason about the value stored in `r` for the ability to treat the unstable read as a right-mover. An implementation of unstable read may exhibit any a subset of the allowed behaviors, including simply performing a conventional read.

[3] The duplicated code in this example could be removed in a language with richer control structures such as `break` statements.

Syntax

$$
\begin{array}{lrcl}
(\textit{Statements}) & s & ::= & \texttt{skip} \mid \texttt{wrong} \mid A \mid s;s \mid \texttt{if}\ C\ s\ \texttt{else}\ s \\
 & & & \mid \texttt{while}\ C\ s \mid f() \mid \texttt{yield} \\
(\textit{Action}) & A & \subseteq & \textit{Tid} \times \textit{Store} \times \textit{Store} \\
(\textit{Thread Identifier}) & t, u & \in & \textit{Tid} = \{1, 2, \ldots\} \\
(\textit{Conditional Action}) & C & ::= & A \diamond A \\
(\textit{Variable Declaration}) & \textit{var} & ::= & x\ \textit{var\_spec} \\
 & x, y, r, m & \in & \textit{Var} \\
(\textit{Function Declaration}) & \textit{fn} & ::= & \textit{fn\_spec}\ \ f()\ \ \{\ s\ \} \\
 & f & \in & \textit{FunctionName} \\
(\textit{Declaration Table}) & D & ::= & \overline{\textit{var} \mid \textit{fn}} \\
 & & & (D \textit{ is left implicit in the semantics for brevity})
\end{array}
$$

Semantics

$$
\begin{array}{lrcl}
(\textit{Store}) & \sigma & \in & \textit{Var} \to \textit{Value} \\
(\textit{State}) & \Sigma & ::= & s_1..s_n \cdot \sigma \\
(\textit{Evaluation Context}) & E & ::= & \bullet \mid E; s
\end{array}
$$

$$\boxed{s \cdot \sigma \to_t s' \cdot \sigma'}$$

$$
\begin{array}{lrcll}
[\text{E-SEQ}] & E[\texttt{skip}; s] \cdot \sigma & \to_t & E[s] \cdot \sigma & \\
[\text{E-YIELD}] & E[\texttt{yield}] \cdot \sigma & \to_t & E[\texttt{skip}] \cdot \sigma & \\
[\text{E-ACTION}] & E[A] \cdot \sigma & \to_t & E[\texttt{skip}] \cdot \sigma' & \text{if } (t, \sigma, \sigma') \in A \\
[\text{E-IF}] & E[\texttt{if}\ (A_1 \diamond A_2)\ s_1\ \texttt{else}\ s_2] \cdot \sigma & \to_t & E[s_i] \cdot \sigma' & \text{if } (t, \sigma, \sigma') \in A_i, \text{ for } i \in 1, 2 \\
[\text{E-WHILE}] & E[\texttt{while}\ C\ s] \cdot \sigma & \to_t & E[\texttt{if}\ C\ (s; \texttt{while}\ C\ s)\ \texttt{else}\ \texttt{skip}] \cdot \sigma & \\
[\text{E-CALL}] & E[f()] \cdot \sigma & \to_t & E[s] \cdot \sigma & \text{if } \textit{fn\_spec}\ \ f()\ \ \{\ s\ \} \in D
\end{array}
$$

$$\boxed{\Sigma \to \Sigma'}$$

$$
[\text{E-STATE}] \quad \frac{s_t \cdot \sigma \to_t s_t' \cdot \sigma'}{s_1..s_t..s_n \cdot \sigma \to s_1..s_t'..s_n \cdot \sigma'}
$$

Figure 6 Mover Logic Language.

We can then express assignments and locking operations as follows. Note that `acquire(`m`)` blocks if the lock is already held, *i.e.* if `\old(`m`)` $\neq 0$. We use the notation $\textit{expr}[x := \texttt{\textbackslash old}(x)]$ to denote *expr* with all occurrences of x replaced by `\old(`x`)`.

$$
\begin{array}{rcl}
\texttt{acquire}(m) & \overset{\text{def}}{=} & \langle \texttt{\textbackslash old}(m) = 0 \wedge m = \textit{tid} \rangle_m \\
\texttt{release}(m) & \overset{\text{def}}{=} & \langle m = 0 \rangle_m \\
x\ \texttt{=}\ \textit{expr} & \overset{\text{def}}{=} & \langle x = \textit{expr}[x := \texttt{\textbackslash old}(x)] \rangle_x
\end{array}
$$

The unstable read $r_{\textit{tid}}$ `~=` x from Section 5.3 may store any value[4] in the local variable $r_{\textit{tid}}$:

$$
r_{\textit{tid}}\ \texttt{\textasciitilde=}\ x \quad \overset{\text{def}}{=} \quad \{\ (\textit{tid}, \sigma, \sigma[r_{\textit{tid}} := v]) \mid v \in \textit{Value}\ \}
$$

[4] In a language with types, this definition can be easily adapted to only store type-correct values into $r_{\textit{tid}}$.

Mover Logic Language includes `if` and `while` statements that condition execution either on whether a Boolean test is true or on whether a store-manipulating operation, such as `cas`, succeeds. To handle these two cases uniformly, we introduce a *conditional action* $C = A_1 \diamond A_2$ where A_1 is an action capturing a true test or successful operation and A_2 is an action capturing a false test or failed operation. For generality, both cases may modify the store and both may be feasible on some pre-states.

We encode any state predicate $B \subseteq \mathit{Store}$ as the conditional action $\{(\mathit{tid}, \sigma, \sigma) \mid \sigma \in B\} \diamond \{(\mathit{tid}, \sigma, \sigma) \mid \sigma \notin B\}$ that distinguishes the true/false cases but never modifies the store. The following illustrates this encoding for the test x `>=` 0.

$$x \texttt{ >= } 0 \quad \stackrel{\text{def}}{=} \quad \{(\mathit{tid}, \sigma, \sigma) \mid \sigma(x) \geq 0\} \diamond \{(\mathit{tid}, \sigma, \sigma) \mid \sigma(x) < 0\}$$

As a more interesting example, we encode `cas` as the following conditional action:

$$\texttt{cas(}x\texttt{,}v\texttt{,}v'\texttt{)} \quad \stackrel{\text{def}}{=} \quad \langle \texttt{\textbackslash old(}x\texttt{)} = v \wedge x = v' \rangle_x \diamond I$$

where the identity action $I = \{\ (t, \sigma, \sigma) \ \mid \ t \in \mathit{Tid} \text{ and } \sigma \in \mathit{Store}\}$. This definition permits `cas` to non-deterministically fail from any pre-state, which enables us to treat failed `cas` operations as both movers [19].

Given $C = A_1 \diamond A_2$, the if statement `if` C s_1 `else` s_2 may either: 1) evaluate the action A_1 and then s_1, or 2) evaluate A_2 and then s_2. The former is the "true" case and the latter is the "false" case, with the desired behavior regardless of whether C encodes a predicate test or a potentially-failing store update. To prevent the if statement from blocking, we require $(A_1 \cup A_2)$ to be total on the state, *i.e.* $\{\ \sigma \mid (t, \sigma, \_) \in (A_1 \cup A_2)\ \} = \mathit{State}$.

The while statement `while` C s behaves similarly. It iterates as long as C succeeds. We may need to test the negation of a conditional action. The negation of $C = A_1 \diamond A_2$, written $!C$, is simply $A_2 \diamond A_1$. The language includes the statement `wrong` to indicate than an error occurred. The statement `assert B` abbreviates `if` B `skip else wrong`. The goal of mover logic is to verify that programs do not go wrong.

Global variable declarations have the form x *var_spec* and are kept in a global declaration table D. Function declarations have the form *fn_spec* $f()$ $\{\ s\ \}$ and are also kept in D. Specifications for globals (*var_spec*) and functions (*fn_spec*) are described in Sections 7 and 8, respectively. For notational simplicity, D is left as an implicit argument to the evaluation judgments. To keep the core language as simple as possible, we elide formal parameters and return values. Instead, parameters and return values are passed in thread-local variables, as described below in Section 7.3.[5]

In our examples, we include types, curly braces, semicolons, and other standard syntactic forms to aid readability.

An execution state

$$\Sigma = s_1..s_n \cdot \sigma$$

consists of sequence of threads $s_1..s_n$ with a shared store σ. The evaluation relation $\Sigma \to \Sigma'$ is based on evaluation contexts $E[\ldots]$, which identify the next statement to be evaluated. A state $\Sigma = s_1..s_n \cdot \sigma$ is *wrong* if any thread is about to execute `wrong`, *i.e.*, if $s_i = E[\texttt{wrong}]$. The semantics demonstrates that `yield` annotations have no effect at run time, but they are used in the mover logic described below.

[5] Extending the language to include function arguments and results is straightforward, but it adds notational complexities that are orthogonal to our core contributions.

7 Mover Logic Effects and Specifications

Mover logic divides the execution of each thread into reducible code sequences that are separated by `yield` statements identifying where thread interference may be observed.

7.1 Effects

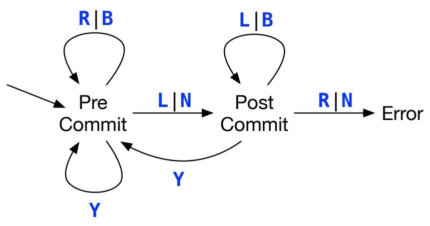
We use a language of effects to reason about reducible code sequences separated by `yield`s:

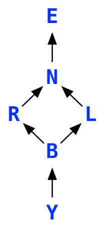
$$e \in \mathit{Effect} \quad ::= \quad \mathtt{Y} \mid \mathtt{R} \mid \mathtt{L} \mid \mathtt{B} \mid \mathtt{N} \mid \mathtt{E}$$

where

- `Y` is the effect of a yield annotation;
- `R` describes right-mover actions;
- `L` describes left-mover actions;
- `B` describes both-mover actions that are both left- and right-movers;
- `N` describes non-mover actions that are neither left- nor right- movers; and
- `E` describes erroneous situations, such as the sequential composition of two non-mover actions without an intervening `yield`, which is not a reducible sequence.

Our strategy for verifying that `yield`s correctly separate reducible sequences is based on the DFA [62] shown below (left). The DFA captures reducible sequences $\mathtt{R}^*[\mathtt{N}]\mathtt{L}^*$ separated by yields `Y`, which resets the DFA to the initial "pre-commit" state on the left to start a new reducible sequence. The first left-mover or non-mover in a reducible sequence is often called the *commit* action and moves us from the pre-commit to the post-commit phase.





From this DFA, we derive the ordering $\mathtt{Y} \sqsubseteq \mathtt{B} \sqsubseteq \mathtt{R}, \mathtt{L} \sqsubseteq \mathtt{N} \sqsubseteq \mathtt{E}$, which is also shown above (right). For example, $\mathtt{R} \sqsubseteq \mathtt{N}$, since for any effect sequences α and β, if $\alpha\,\mathtt{N}\,\beta$ is accepted by this DFA, the $\alpha\,\mathtt{R}\,\beta$ is also accepted. We define a standard join operation $\sqcup$ via this ordering.

We also define sequential composition $e_1; e_2$ and iterative closure e^*, as in [62]. For example, $\mathtt{R};\mathtt{L} = \mathtt{N}$ since to show $\alpha\,\mathtt{R}\,\mathtt{L}\,\beta$ is accepted by the DFA it is sufficient to show that $\alpha\,\mathtt{N}\,\beta$ is accepted. Conversely, $\mathtt{N};\mathtt{N} = \mathtt{E}$ (error), since $\alpha\,\mathtt{N}\,\mathtt{N}\,\beta$ is never accepted by this DFA.

$e_1; e_2$	Y	B	R	L	N	E
Y	Y	Y	Y	L	L	E
B	Y	B	R	L	N	E
R	R	R	R	N	N	E
L	Y	L	E	L	E	E
N	R	N	E	N	E	E
E	E	E	E	E	E	E

e	e^*
Y	Y
B	B
R	R
L	L
N	E
E	E

7.2 Mover Specifications

In mover logic, the verification of a thread *tid* is performed in the context of a mover specification describing how each program action A starting in the store σ commutes with steps of other threads. Thus, mover specifications M have the type

$$M : \mathit{Action} \times \mathit{Tid} \times \mathit{Store} \to \mathit{Effect} \setminus \{\mathtt{Y}\}$$

For example, if action A is a local computation that only accesses thread-local variables, we would naturally have

$$M(A, \mathit{tid}, \sigma) = \mathtt{B}$$

Alternatively, if a global variable `x` is protected by a lock `m`, the write action `x = ` *expr* might have the mover specification

$$M(\mathtt{x = }\ \mathit{expr}, \mathit{tid}, \sigma) = \begin{cases} \mathtt{B} & \text{if } \sigma(\mathtt{m}) = \mathit{tid} \\ \mathtt{E} & \text{otherwise} \end{cases}$$

indicating that the write is a both-mover only if thread *tid* holds lock `m`. Otherwise, it is an error. We assume that *expr* only accesses local variables, and that $M(A, \mathit{tid}, \sigma)$ is never `Y` since actions do not yield.

We write mover specifications in the source code using the following notation, which is inspired by earlier reduction-based verifiers [30, 19, 21]:

$$\begin{array}{rcl} \mathit{var\_spec} & ::= & \mathit{var\_clause}^* \\ \mathit{var\_clause} & ::= & \mathtt{read}\ e\ \mathtt{if}\ P \mid \mathtt{write}\ e\ \mathtt{if}\ P \end{array}$$

where $P \subseteq \mathit{Tid} \times \mathit{Store} \times \mathit{Store}$ is a two-store predicate describing the pre-store and post-store of the access to x in question. Further, P can depend on the current thread identifier *tid*. Similar to actions, we write these predicates as formulae in which `\old(`y`)` and y to refer to the values of y in the pre-store and post-store, respectively. To determine the mover effect of a variable access, we evaluate the specification clauses in order and take the effect of the first case where the condition P is satisfied. If no clauses apply, the access has the error effect `E`. More formally, given the specification for a variable `x` in the source code, we collect the sequence of clauses for reads and writes separately and then create the mover specification M for `x` as follows:

$$\left[\!\!\left[\begin{array}{c} \mathtt{read}\ e_1\ \mathtt{if}\ P_1 \\ \vdots \\ \mathtt{read}\ e_n\ \mathtt{if}\ P_n \end{array} \right]\!\!\right] \implies M(\mathtt{r}_{\mathit{tid}} = \mathtt{x}, \mathit{tid}, \sigma) = \begin{cases} e_1 & \text{if } P_1(\mathit{tid}, \sigma, \sigma) \\ \vdots & \vdots \\ e_n & \text{if } P_n(\mathit{tid}, \sigma, \sigma) \\ \mathtt{E} & \text{otherwise} \end{cases}$$

$$\left[\!\!\left[\begin{array}{c} \mathtt{write}\ e_1\ \mathtt{if}\ P_1 \\ \vdots \\ \mathtt{write}\ e_n\ \mathtt{if}\ P_n \end{array} \right]\!\!\right] \implies M(\mathtt{x} = \mathit{expr}, \mathit{tid}, \sigma) = \begin{cases} e_1 & \text{if } P_1(\mathit{tid}, \sigma, \sigma[\mathtt{x} := \sigma(\mathit{expr})]) \\ \vdots & \vdots \\ e_n & \text{if } P_n(\mathit{tid}, \sigma, \sigma[\mathtt{x} := \sigma(\mathit{expr})]) \\ \mathtt{E} & \text{otherwise} \end{cases}$$

where $\mathtt{r}_{\mathit{tid}}$ is a local variable, *expr* only accesses thread-local variables, $\sigma(\mathit{expr})$ is the result of evaluating *expr* in the store σ, and the cases for M are evaluated in the order listed.

Counter Library

```
    int x   both-mover if m == tid
    lock m  write right-mover
             if \old(m) == 0 && m == tid
            write left-mover
             if \old(m) == tid && m == 0

    atomic    non-mover
    requires  true
    ensures   x == \old(x) + arg1_tid
    ensures   result_tid == x
    add() {
R      ⟨\old(m) == 0 ∧ m == tid⟩_m
B      r_tid = x;
B      r_tid = r_tid + arg1_tid;
B      x = 1;
B      x = r_tid;
L      ⟨m == 0⟩_m;
B      result_tid = r_tid;
    }
```

Client

```
    relies     even(x)
    guarantees even(x)
    requires   even(x)
    ensures    even(x)
    client() {
         // even(x)
B      arg1_tid = 2;
         // even(x) && arg1_tid == 2
N      add_tid();
         // even(x)
Y      yield;
         // even(x)
B      arg1_tid = 2;
         // even(x) && arg1_tid == 2
N      add();
         // even(x) && even(result_tid)
B      u_tid = result_tid;
         // even(x) && even(u_tid)
B      if even(u_tid) skip else wrong;
         // even(x)
Y      yield;
         // even(x)
    }
```

Initial State Σ

```
(yield; client()).(yield; client())·[x := 0, m := 0]
```

Figure 7 The example from Figure 3 (right) expressed in Mover Logic Language.

The declaration for a global variable `x` protected by a lock `m` is thus written as

```
int x   read   both-mover if m == tid
        write  both-mover if m == tid
```

where **both-mover** is syntactic sugar for the effect B. (Similarly, we use **left -mover** for L, and so on.) In our examples, we abbreviate these identical read and write cases as follows.

```
int x   both-mover if m == tid
```

7.3 Motivating Example, Revisited

Figure 7 expresses our motivating example from Figure 3 (right) in our Mover Logic Language. As mentioned earlier, an access to a thread-local variable r actually accesses a (global) variable r_{tid} that is reserved for use by thread *tid*. We use thread-local variables to encode function arguments and results. The fork statements are converted into parallel threads in the initial state Σ. We insert a `yield` at the start of each thread in Σ so that the initial state is well-formed under the non-preemptive semantics we introduce in our formal development.

Given this mover specification, mover logic successfully verifies this code. Figure 7 also demonstrates the reasoning carried out by mover logic. The left margin shows the effect of each action and groups those effects into reducible sequences. The `add()` function is a single reducible sequence, ensuring that we may treat it as atomic. The `client()` function consists of multiple reducible sequences separated by `yield`s. We also show invariants demonstrating that `client()` is correct in comments at each program point.[6]

7.4 Additional Mover Specification Examples

Figure 3 (right) showed how mover specifications can capture the synchronization/commuting behavior of lock acquires, lock releases, and lock-protected variable accesses. Our mover specifications are inspired by the Anchor verifier, which used mover specifications to capture many synchronization idioms [19, 1].[7]

To illustrate how mover specifications capture more complex synchronization disciplines, suppose the variable `y` is *write-protected* by a lock `m`. That is, lock `m` must be held for all writes to `y` but not necessarily held for reads. Consequently, `y` should be declared volatile if the code is run under a weak memory model. Writes to `y` are non-movers (due to concurrent reads); lock-protected reads are both-movers (because there can be no concurrent writes); and reads without holding the lock are non-movers (due to concurrent writes). Mover specifications capture this synchronization discipline concisely as follows, where the last clause applies only when `m` is not `tid`.

```
int y write non-mover  if m == tid
      read  both-mover if m == tid
      read  non-mover
```

The FastTrack dynamic race detector [18, 57] uses a combination of lock-protected and write-protected disciplines to synchronize accesses to some array pointers. We illustrate that discipline for an array pointer `vc`: initially, a `flag` is false and the pointer `vc` is guarded by `lock`; when `flag` becomes true, `vc` becomes write-guarded by `lock`. The mover specification for this discipline is captured by the first four lines in the specification for `vc`:

```
int vc[]        both-mover if !flag && lock == tid
          write non-mover  if  flag && lock == tid
          read  both-mover if  flag && lock == tid
          read  non-mover  if  flag
          [i]          both-mover if !flag && lock == tid
          [i] read     both-mover if  flag && (lock == tid || tid == i)
          [i] write    both-mover if  flag && (lock == tid && tid == i)
```

This idiom enables the algorithm to avoid using a lock to protected all accesses to `vc` but still replace `vc` with a larger array when necessary. The last three lines capture the synchronization discipline for accessing the array entry `vc[i]`, where we use the extended notation "`[i]` *var_clause*" to describe the synchronization cases for actions that access `vc[i]`. That entry is also initially guarded by `lock` when `flag` is false; when `flag` becomes true, the entry `vc[i]` can only be written by thread `i` while holding `lock`, read by any thread while holding the lock, or read by thread `i` without holding the lock. These reads and writes are all both-movers. These rules prevent all conflicting reads and writes, and thus all accesses to `vc[i]` are both-movers under this synchronization discipline.

[6] In this example, the rely assumption `even(x)` is sufficient for reasoning about `yield` points. In code where live ranges for local variables span yield points, we would add to the rely assumptions the requirement that one thread does not change another thread's local variables.

[7] Our syntax for mover specifications is a syntactic variant of the Anchor syntax. In essence, our specifications are sequential *var_clauses*, whereas Anchor combines these clauses into a single binary decision tree using the syntax *bool_expr* ? *mover_spec* : *mover_spec*.

As a final example, consider a concurrent hashtable consisting of a `table` array and a `locks` array, which has length `N`. The entry `table[i]` is protected by `locks[i % N]`. The `table` reference itself may change when, for example, `table` is replaced with a larger array. To ensure such changes are done without interference, a write to `table` is permitted only when a thread holds *all* locks. In contrast, `table` can be read by a thread holding *any* lock. All such reads and writes are both-movers, as captured by the following mover specification:

```
Entry table[] write both-mover if ∀i ∈ [0,N). locks[i] == tid
              read  both-mover if ∃i ∈ [0,N). locks[i] == tid
              [i]   both-mover if locks[i % N] == tid
```

As illustrated in the previous two examples from the Anchor verifier [19], mover specifications can naturally capture synchronization disciplines that vary with the current program state.

A final example comes from the common iterative parallel algorithm pattern in which a synchronization barrier is used to divide the computation into a series of phases. In the even phases, the main thread (with `tid` $= 0$) updates shared data structures, and in odd phases, worker threads concurrently read data from those structures, as specified below.

```
int z  read both-mover if phase % 2 == 1
            both-mover if phase % 2 == 0 && tid == 0
```

8 Mover Logic

In this section, we show the proof rules for how mover logic handles statements (Section 8.1); function definitions, calls, and specifications (Sections 8.2–8.3); and run-time states (Section 8.4).

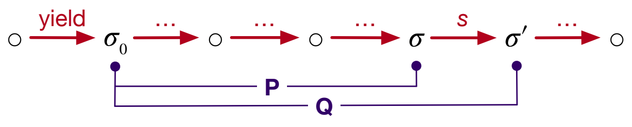
8.1 Mover Logic

Mover logic is defined via the judgments in Figures 8 and 9. The main judgment

$$R; G \vdash s : P \to Q \cdot e$$

verifies that, when starting from a store satisfying the precondition P, the statement s terminates only in stores satisfying the postcondition Q (*i.e.* partial correctness). In addition, the judgment uses the mover specification M to verify that s consists of reducible sequences separated by `yield`s. At each yield point, the rely assumption $R \subseteq \mathit{Tid} \times \mathit{Store} \times \mathit{Store}$ is used to model potential interference from other threads. Conversely, the thread guarantee $G \subseteq \mathit{Tid} \times \mathit{Store} \times \mathit{Store}$ summarizes the behavior of each reducible code sequence between two yield points in s. The effect e summarizes how s commutes with steps of other threads.

In the rules, the precondition P can refer to the value of variable `x` in the initial store σ_0 of the current reducible code sequence via the notation `\old(x)`. Thus P is a two-store relation $P \subseteq \mathit{Tid} \times \mathit{Store} \times \mathit{Store}$ relating that initial store σ_0 to the pre-store σ for the execution of s. We show that requirement visually in the following trace, where $(\mathit{tid}, \sigma_0, \sigma) \in P$.



One-Store and Two-Store Predicates and Supporting Definitions

$$\begin{aligned} S, T &\subseteq \mathit{Tid} \times \mathit{Store} \\ R, G, P, Q, A &\subseteq \mathit{Tid} \times \mathit{Store} \times \mathit{Store} \end{aligned}$$

$$\begin{aligned} \mathit{Two}(S) &\overset{\text{def}}{=} \{\ (t, \sigma, \sigma) \mid (t, \sigma) \in S\ \} \\ \mathit{Post}(P) &\overset{\text{def}}{=} \{\ (t, \sigma) \mid (t, \_, \sigma) \in P\ \} \\ I &\overset{\text{def}}{=} \{\ (t, \sigma, \sigma) \mid t \in \mathit{Tid}, \sigma \in \mathit{Store}\ \} \end{aligned}
\qquad
\begin{aligned} P; A &\overset{\text{def}}{=} \left\{ (t, \sigma, \sigma'') \;\middle|\; \begin{array}{l} (t, \sigma, \sigma') \in P \text{ and} \\ (t, \sigma', \sigma'') \in A \end{array} \right\} \\ \mathit{Yield}(P, R) &\overset{\text{def}}{=} \left\{ (t, \sigma', \sigma') \;\middle|\; \begin{array}{l} (t, \_, \sigma) \in P \text{ and} \\ (t, \sigma, \sigma') \in R^* \end{array} \right\} \end{aligned}$$

Mover Logic Proof Rules

$$\boxed{R; G \vdash s : P \to Q \cdot e}$$

[M-ACTION]
$$\frac{\begin{array}{c} M(A, P) = e \\ e \sqsubseteq \mathtt{L} \Rightarrow A \text{ is total} \end{array}}{R; G \vdash A : P \to (P; A) \cdot e}$$

[M-SEQ]
$$\frac{\begin{array}{c} R; G \vdash s_1 : \ P \ \to Q_1 \cdot e_1 \\ R; G \vdash s_2 : Q_1 \to Q_2 \cdot e_2 \end{array}}{R; G \vdash s_1; s_2 : P \to Q_2 \cdot (e_1; e_2)}$$

[M-IF]
$$\frac{\begin{array}{c} R; G \vdash s_1 : P; A_1 \to Q \cdot e_1 \\ R; G \vdash s_2 : P; A_2 \to Q \cdot e_2 \\ e = (M(A_1, P); e_1) \sqcup (M(A_2; P); e_2) \end{array}}{R; G \vdash \mathtt{if}\ (A_1 \diamond A_2)\ s_1\ \mathtt{else}\ s_2 : P \to Q \cdot e}$$

[M-WHILE]
$$\frac{\begin{array}{c} R; G \vdash s : P; A_1 \to P \cdot e_1 \\ e = (M(A_1, P); e_1)^*; M(A_2, P) \\ e \not\sqsubseteq \mathtt{L} \end{array}}{R; G \vdash \mathtt{while}\ (A_1 \diamond A_2)\ s : P \to P; A_2 \cdot e}$$

[M-SKIP]
$$\frac{}{R; G \vdash \mathtt{skip} : P \to P \cdot \mathtt{B}}$$

[M-WRONG]
$$\frac{}{R; G \vdash \mathtt{wrong} : \emptyset \to \emptyset \cdot \mathtt{B}}$$

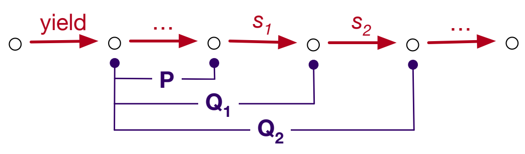
[M-CONSEQ]
$$\frac{\begin{array}{c} P \Rightarrow P_1 \qquad R \Rightarrow R_1 \\ Q_1 \Rightarrow Q \qquad G_1 \Rightarrow G \qquad e_1 \sqsubseteq e \\ R_1; G_1 \vdash s : P_1 \to Q_1 \cdot e_1 \end{array}}{R; G \vdash s : P \to Q \cdot e}$$

[M-YIELD]
$$\frac{\begin{array}{c} P \Rightarrow G \\ Q = \mathit{Yield}(P, R) \end{array}}{R; G \vdash \mathtt{yield} : P \to Q \cdot \mathtt{Y}}$$

Figure 8 Mover logic proof rules and supporting definitions.

The two-store postcondition $Q \subseteq \mathit{Tid} \times \mathit{Store} \times \mathit{Store}$ relates σ_0 to the post-store σ' of s.

Many of the mover logic rules are extensions of Hoare logic incorporating reduction effects. For example, the rule [M-SEQ] states that a sequential composition $(s_1; s_2)$ commutes as $e_1; e_2$, the sequential composition of the effects of its sub-statements, and that the precondition and postcondition are related as follows:



The rule [M-SKIP] indicates that `skip` has no effect, so its precondition and postcondition are identical. The rule [M-WRONG] verifies that `wrong` is never executed via the unsatisfiable precondition $\emptyset$. That is, this rule rejects any program that may execute `wrong` from any state.

Function Specification Syntax

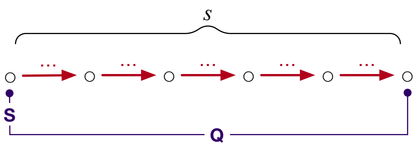
$$\begin{array}{rcrl} \mathit{fn\_spec} & ::= & \textbf{atomic}\ e & \textbf{requires}\ S\ \textbf{ensures}\ Q \\ & \mid & \textbf{relies}\ R\ \textbf{guarantees}\ G & \textbf{requires}\ S\ \textbf{ensures}\ T \end{array}$$

Proof Rules for Function Definitions and Calls

$\boxed{\vdash \mathit{fn}}$

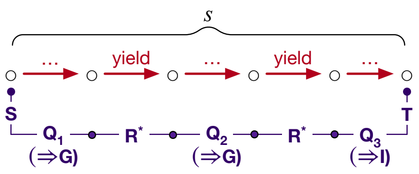
[M-DEF-ATOMIC]

$$\frac{f()\ \text{is not (directly or indirectly) recursive} \qquad \emptyset;\emptyset \vdash s : \mathit{Two}(S) \to Q \cdot e}{\vdash \textbf{atomic}\ e\ \ \textbf{requires}\ S\ \textbf{ensures}\ Q\quad f()\ \{\ s\ \}}$$



[M-DEF-NON-ATOMIC]

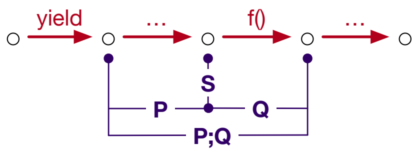
$$\frac{R;G \vdash s : \mathit{Two}(S) \to \mathit{Two}(T) \cdot \mathtt{R} \qquad G \neq \emptyset}{\vdash \textbf{relies}\ R\ \textbf{guarantees}\ G\ \ \textbf{requires}\ S\ \textbf{ensures}\ T\quad f()\ \{\ s\ \}}$$



$\boxed{R;G \vdash s : P \to Q \cdot e}$

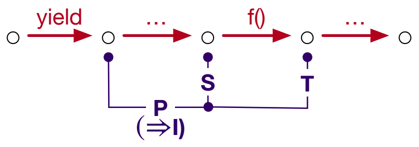
[M-CALL-ATOMIC]

$$\frac{\textbf{atomic}\ e\ \ \textbf{requires}\ S\ \textbf{ensures}\ Q\quad f()\ \{\ s\ \} \in D \qquad \mathit{Post}(P) \Rightarrow S}{R;G \vdash f() : P \to (P;Q) \cdot e}$$



[M-CALL-NON-ATOMIC]

$$\frac{\textbf{relies}\ R\ \textbf{guarantees}\ G\ \ \textbf{requires}\ S\ \textbf{ensures}\ T\quad f()\ \{\ s\ \} \in D}{R;G \vdash f() : \mathit{Two}(S) \to \mathit{Two}(T) \cdot \mathtt{R}}$$



Verification of States

$\boxed{\vdash \Sigma}$

[M-STATE]

$$\frac{\begin{array}{c} \forall \mathit{fn} \in D.\ \vdash \mathit{fn} \qquad M\ \text{is valid} \qquad I \Rightarrow G \\ \forall t \in \mathit{Tid}.\ \left[\begin{array}{l} R;G \vdash s_t : P_t \to Q_t \cdot e_t\ \text{and}\ e_t \neq \mathtt{E}\ \text{and}\ Q_t \Rightarrow G \\ \text{and}\ s_t\ \text{is yielding and}\ (t,\sigma,\sigma) \in P_t \end{array}\right] \\ \forall t,u \in \mathit{Tid}.\ t \neq u \Rightarrow (G[\mathit{tid} := t] \Rightarrow R[\mathit{tid} := u]) \end{array}}{\vdash s_1..s_n \cdot \sigma}$$

Figure 9 Mover logic proof rules for function definition, calls, and run-time states.

The rule [M-ACTION] computes the effect of action A from states σ satisfying the current precondition P. That rule uses the function to compute this effect:

$$M(A,P) \overset{\text{def}}{=} \bigsqcup_{(t,\_,\sigma)\in P} M(A,t,\sigma)$$

(Note that we are overloading M here.) The postcondition of A is then the precondition P sequentially composed with the action A, *i.e.* $P;A$. A key technical requirement of the reduction theorem is that once an atomic block $\texttt{R}^*[\texttt{N}]\texttt{L}^*$ enters its post-commit (or left-mover part), then it must terminate. It cannot block or diverge [24].[8] Hence, we require that A is total if it is a left-mover. We place similar restrictions on loops.

The rule [M-IF] requires both the true case $(A_1; s_1)$ and the false case $(A_2; s_2)$ to have the same post-condition Q. The effect e is the maximal effect of executing either A_1 followed by s_1 or A_2 followed by s_2. The rule [M-WHILE] for `while` $A_1 \diamond A_2\ s$ checks that a successful test followed by the body preserves precondition P, which functions as a loop invariant. The postcondition of the loop is the postcondition of A_2 given the precondition P. The effect of a loop is the iterative closure of the effect of one iteration sequentially composed with the effect of the loop-terminating test A_2.

Consider the loop in `spin_lock()` in Figure 5. The test `!cas(l,0,tid)` is the conditional action $I \diamond \langle \texttt{\textbackslash old(l)} = 0 \wedge \texttt{l} = \texttt{tid}\rangle_1$ and the loop body is `skip`. Since $P; I = P$, rule [M-SKIP] concludes that $R; G \vdash \texttt{skip} : P \to P \cdot \texttt{B}$. Further, $M(I,P) = \texttt{B}$, because that action accesses no global variables, and the specification for `l` indicates that $M(\langle \texttt{\textbackslash old(l)} = 0 \wedge \texttt{l} = \texttt{tid}\rangle_1, P) = \texttt{R}$. Thus, $e = (\texttt{B};\texttt{B})^*;\texttt{R} = \texttt{R}$. Also, the postcondition $P; A_2$ for the loop simplifies to the expected $P[\texttt{l} := \texttt{tid}]$. To ensure the left-mover termination requirement, rule [M-WHILE] requires that $e \not\sqsubseteq \texttt{L}$. That is, the post-commit part of a reducible sequence cannot contain loops.

The rule [M-YIELD] for `yield` first checks that the thread guarantee G includes all possible behaviors P of the reducible sequence preceding the `yield` via the antecedent $P \Rightarrow G$. The reducible sequence following the `yield` starts with postcondition $Q = \mathit{Yield}(P,R)$ which incorporates repeated thread interference from other threads via the iterated rely assumption R^* and then resets each `\old(x)` value to be the current value of `x` at the start of the new reducible sequence.

The rule [M-CONSEQ] extends the consequence rule of RG logic to reduction effects.

8.2 Atomic Functions

Mover logic supports both atomic and non-atomic functions. An atomic function is one whose code body is reducible (*i.e.*, no `yield` statements) and has the following form:

$$\begin{array}{l}\textbf{atomic}\ e \\ \textbf{requires}\ S\ \textbf{ensures}\ Q \quad f()\ \{\ s\ \}\end{array}$$

(We elide e in the surface syntax when it is $\texttt{N}$, as in Figure 3 (right)). The precondition $S \subseteq \mathit{Tid} \times \mathit{Store}$ describes valid initial stores for the function call and must be established by the caller. The post condition $Q \subseteq \mathit{Tid} \times \mathit{Store} \times \mathit{Store}$ describes possible final stores, and it may refer to values of variables on function entry using the `\old(x)` notation. Since s is

[8] To motivate this requirement consider the program `(x = 1; while (true) skip; yield) || (assert x != 1)`. This program can go wrong because the first thread writes 1 to `x`. However, the reducible block containing that write never terminates after performing that write, and that write is not included in the thread guarantee G. Thus, we require that once a reducible block commits, it must terminate.

atomic and `yield`-free, we elide the rely and guarantee components from atomic function specifications. We require atomic functions to be non-recursive to facilitate the "left-mover terminates" requirement mentioned above.

To ensure that the function body s conforms to the function's specification, rule [M-DEF-ATOMIC] in Figure 9 first converts S into the two-store precondition $\mathit{Two}(S)$ (in which `\old(x) = x` for all variables `x`) and then verifies the function body s with respect to that precondition. We use the guarantee $\emptyset$ to enforce that s is indeed yield-free. (Rule [M-YIELD] will always fail if G is $\emptyset$, provided that the `yield` is actually reachable, *i.e.* if $P \neq \emptyset$).

The rule [M-CALL-ATOMIC] for a corresponding call to $f()$ retrieves the above specification from the declaration table D and then ensures that the precondition P at the call site implies the callee's precondition S. That rule uses $\mathit{Post}(P)$ to first convert P into a one-state predicate. The postcondition $(P; Q)$ combines the call precondition P with the two-store postcondition Q of the callee, as illustrated in the trace to the right of the rule.

8.3 Non-Atomic Functions

Non-atomic function definitions have the following form:

requires R **guarantees** G
requires S **ensures** T $\quad f()\ \{\ s\ \}$

We include thread rely R and guarantee G components in these function specifications since non-atomic function may include `yield` points where thread interference may occur. For simplicity, we require that non-atomic function calls and returns happen at the start of a reducible sequence. Consequently, the precondition $S \subseteq \mathit{Tid} \times \mathit{Store}$ and postcondition $T \subseteq \mathit{Tid} \times \mathit{Store}$ are both one-store predicates since there is no need to summarize the preceding reducible sequence.

The rule [M-DEF-NON-ATOMIC] checks that the function body s runs from the precondition $\mathit{Two}(S)$, possibly via multiple reducible sequences separated by `yield`s, to terminate after a final `yield` s in a store satisfying T. Those requirements are enforced by using $\mathit{Two}(T)$ as the postcondition for s. Further, the body s should end in a yield, which from the definition of $e_1; e_2$ entails that the effect of s is at most `R`. At a call site, the rule [M-CALL-NON-ATOMIC] requires that the current reducible sequence is trivial/empty and meets the function's one-store precondition S by requiring the precondition $\mathit{Two}(S)$ prior to the call. The rule also converts the function's one-store postcondition T to the two-store predicate $\mathit{Two}(T)$.

8.4 Verifying States

We now define the verification judgment $\vdash \Sigma$ to verify program states $\Sigma = s_1..s_n \cdot \sigma$. The rule [M-STATE] for this judgment in Figure 9 ensures that:

- each thread s_t verifies from a precondition P_t that includes the initial store σ;
- any pending behavior Q_t at thread termination is published to G;
- the thread guarantee G is reflexive;
- the guarantee of each thread is contained in the rely assumption of every other thread;
- each function definition in the global table D is verifiable; and
- that all threads start with a `yield` statement (to simplify the correctness proof).

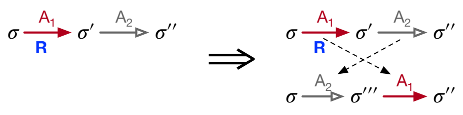
A mover specification M makes claims about how steps of one thread commute with respect to steps of other threads, and mover logic needs to ensure that those claims are correct. Specifically, we define a mover specification to be *valid* if:

1. Right-moving actions can be moved later in a trace without changing the final store.
2. Left-moving actions can be moved earlier in a trace without changing the final store.
3. An action by one thread cannot change the effect of an action in another thread.
4. An action by one thread cannot cause a left-moving action in another thread to block.

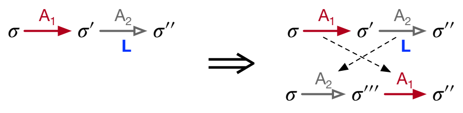
We formalize these validity requirements as follows:

▶ **Definition 1** (Validity). *M is* valid *if the following four conditions hold for all threads $t \neq u$ and $A_1, A_2, \sigma, \sigma'$:*

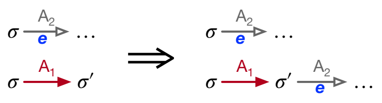
*(1) if $M(A_1, t, \sigma) \sqsubseteq$ **R** and $(t, \sigma, \sigma') \in A_1$ and $M(A_2, u, \sigma') \sqsubseteq$ **N** and $(u, \sigma', \sigma'') \in A_2$, then there exists σ''' such that $(u, \sigma, \sigma''') \in A_2$ and $(t, \sigma''', \sigma'') \in A_1$.*



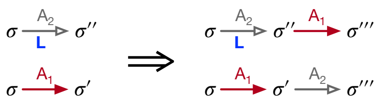
*(2) if $M(A_1, t, \sigma) \sqsubseteq$ **N** and $(t, \sigma, \sigma') \in A_1$ and $M(A_2, u, \sigma') \sqsubseteq$ **L** and $(u, \sigma', \sigma'') \in A_2$, then there exists σ''' such that $(u, \sigma, \sigma''') \in A_2$ and $(t, \sigma''', \sigma'') \in A_1$.*



*(3) if $M(A_1, t, \sigma) \sqsubseteq$ **N** and $(t, \sigma, \sigma') \in A_1$ and $M(A_2, u, \sigma) = e$ for some e, then $M(A_2, u, \sigma') = e$.*



*(4) if $M(A_1, t, \sigma) \sqsubseteq$ **N** and $(t, \sigma, \sigma') \in A_1$ and $M(A_2, u, \sigma) \sqsubseteq$ **L** and $(u, \sigma, \sigma'') \in A_2$, then there exists σ''' such that $(u, \sigma', \sigma''') \in A_2$ and $(t, \sigma'', \sigma''') \in A_1$.*



8.5 Correctness

The central correctness theorem for mover logic is that verified programs do not go wrong by, for example, failing an assertion.

▶ **Theorem 2** (Soundness). *If $\vdash \Sigma$ then Σ does not go wrong.*

The proof appears in full in the extended version of this paper [20]. The basic structure is as follows.

1. We first develop an instrumented semantics that enforces the mover specification M and also that each thread consists of reducible sequences separated by yields.
2. In addition to the usual preemptive scheduler, we also develop a non-preemptive scheduler for the instrumented semantics that context switches only at `yield`s.
3. We show that the instrumented semantics under the preemptive scheduler behaves the same as the standard semantics except that it may go wrong more often.
4. We use a reduction theorem to show that programs exhibit the same behavior under the preemptive and non-preemptive instrumented semantics.
5. Finally, we use a preservation argument [58] to show that verified programs do not go wrong under the non-preemptive instrumented semantics.
6. The steps above then imply that verified programs do not go wrong under the preemptive standard semantics.

9 Related Work

Modular Reasoning

Concurrent software verification introduces a number of scalability challenges that require a synthesis of various notions of modularity or abstraction to address. For example, *procedure-modular* reasoning tackles large code bases by verifying each procedure with respect to a specification of other procedures in the system. Rely-guarantee logic [33] augments procedure-modular reasoning with a notion of *thread-modular* reasoning that accommodates multiple threads by verifying each thread with respect to a specification of other threads in the system. As demonstrated in Section 2, systems like RG logic that support procedure-modular and thread-modular reasoning have great potential, but they are limited by entanglement between library and client specifications.

To address that limitation, mover logic augments procedure-modular and thread-modular reasoning with Lipton's theory of reduction [41]. This complementary form of "interleaving" modularity limits the number of interleavings that must be considered and enables more precise procedure specifications for atomic functions.

In other work, separation logic combines procedure-modular reasoning with a notion of *heap-modular* reasoning [47, 49], which enables verification of sub-goals while ignoring irrelevant heap objects. Separation logic has been the foundation for a variety of verification tools [3, 32, 44]. Concurrent separation logics including, for example [53, 46, 5, 52], extend those ideas to a concurrent setting. While initially focused on noninterference via disjoint access and read-only sharing, later work [14, 13] supports more tightly-coupled threads.

Much of the work on concurrent separation logic focuses on *resources* (*e.g.*, heap locations) and on ensuring threads access disjoint resources (hence ensuring noninterference). In contrast, mover logic focuses on commuting *actions*.

Concurrent separation logic and mover logic also differ in where thread interference specifications are placed. Concurrent separation logic conveniently merges interference (or resource footprint) specifications into each method's precondition, thus enabling the logic to capture sophisticated resource usage idioms in a concise and elegant manner. Deny-guarantee reasoning [14] extends concurrent separation logic to focus more on actions rather than resources. In particular, a method's precondition can include an "action map" specifying what actions the method (and its concurrent threads) may perform. This action map is analogous to our mover specifications. Several projects employ permissions or ownership, similar to separation logic, to reason about which memory locations are available to different threads. These include Viper [43] and VerCors [4]. These systems do not support reduction.

An important topic of for future study is how to extend mover logic with a notion of heap modularity, perhaps similar to the core ideas of concurrent separation logic or dynamic frames [2, 51, 35]. This body of work may also provide insight into how to develop a compositional semantics based on mover logic.

Reduction-based Techniques

QED [15] is a program calculus and verification procedure for concurrent programs. It utilizes iterative reduction and abstraction refinement to increase the size of the blocks that can be considered serializable regions (at the abstract level). That approach has been shown to be quite successful for verifying complex concurrent code and has inspired a number of subsequent verification tools described below. Mover logic is a complementary approach in that the combination of RG reasoning and reduction enables direct verification of code with `yield` points, without the need to create layers of abstractions. As part of that, mover logic supports specifying and reasoning about functions that are not atomic, which is not supported in QED. We also note that QED checks the commutativity properties of an action via a pairwise check with all other actions in the code, whereas mover logic uses the mover specification validity check for that purpose.

Several more recent verification tools utilize the same approach of writing a series of programs related by refinement, abstraction, and reduction. These include the CIVL verifier [30, 38, 40, 36, 37, 39] and the Armada verifier [42]. They are capable of handling sophisticated concurrent code, but do require the programmer to write and maintain multiple versions of the source code. The correctness arguments for these tools have typically been based on monolithic proofs.

Calvin-R [25] developed a number of early ideas related to reduction and thread-modular reasoning. The Anchor verifier [19] builds on ideas behind Calvin-R and CIVL to create a verification technique supporting an executable, object-oriented target language, a variety of synchronization primitives, and a new notation for specifying the interference between threads that is the foundation for our mover specifications. While effective at some verification tasks, Anchor's correctness arguments are also challenging to understand and build upon. Further, Anchor is inherently limited to small programs because it inlines nested calls during verification, with no mechanism for procedure-modular reasoning. Mover logic may provide a useful foundation for a procedure-modular extension of Anchor.

The difficulty in assessing the strengths and weaknesses of the tools mentioned above without a robust underlying logic capturing what they do inspired this work. Mover logic may provide such a foundation, detached from any particular full-scale implementation, that it is accessible, general, and extensible. We hope implementations based on mover logic will follow, as the logic clarifies exactly what conditions must be met in reduction-based verifiers that attempt to integrate modular reasoning in the presence of interference.

Coq-based Techniques

Complementary approaches develop proof frameworks for verifying concurrent programs in Coq [12]. For example, CCAL [28] provides a compositional semantic model for composing and verifying the correctness of multithreaded components. CCAL focuses on only rely-guarantee reasoning [33] and not reduction. CSpec [7] is a Coq library for verifying concurrent systems modeled in Coq [12] using movers and reduction. While highly expressive, particularly because additional proof techniques can be added as additional Coq code, users must write significant Coq code for both specifications and proofs to use such a system. We have focused on a logic more amenable to fully automatic reasoning. Iris [34] uses higher-order separation logic to verify correctness of higher-order imperative programs.

Model Checking

An orthogonal approach to software verification utilizes explicit state, exhaustive model checking. Such approaches have lower programmer overhead than other techniques, but they are non-modular [16, 10, 11]. Specialized techniques, including reduction [29] and partial-order methods [27, 26, 48], have been used to limit state-space explosion while checking concurrent programs. A variety of concurrent software model checkers [8, 59, 45] have demonstrated the potential of these approaches in constrained settings.

10 Summary

Over the last two decades, several promising multithreaded program verifiers have leveraged reduction to verify sophisticated concurrent code including non-blocking algorithms, dynamic data race detectors, and garbage collectors by leveraging precise, reusable specifications for atomic functions. The reasoning used by these verifiers, including the notion of which programs are verifiable, and why the verification process is sound, is unfortunately rather complex. In contrast, Hoare logic [31] provides an accessible foundation for sequential verifiers, and RG logic [33] provides a similar foundation for some multithreaded verifiers.

In developing mover logic, we aim to facilitate future research on reduction-based verification. Mover logic provides a declarative and formal explanation of reduction-based verification, making it easier to understand which programs are verifiable, or not, and why; which functions can be specified as atomic; what atomic and non-atomic function specifications mean; which code blocks are reducible; where yield annotations are required, etc. The correctness proof for a reduction-based verifier need only show that the verifier follows the rules of mover logic, a significant simplification over existing proof techniques.

We hope that mover logic inspires the development of more expressive reduction-based logics and verification tools, potentially supporting features such as objects, data abstraction, dynamic allocation, dynamic thread creation, and precise frame conditions [2, 51, 35].

References

1 The Anchor verifier. Accessed: March 30, 2024. URL: `http://www.anchor-verifier.com/`.

2 Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP*, volume 5142 of *Lecture Notes in Computer Science*, pages 387–411. Springer, 2008.

3 Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.

4 Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. The vercors tool set: Verification of parallel and concurrent software. In *IFM*, volume 10510 of *Lecture Notes in Computer Science*, pages 102–110. Springer, 2017.

5 Stephen Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007.

6 Pavol Cerný, Edmund M. Clarke, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, Roopsha Samanta, and Thorsten Tarrach. From non-preemptive to preemptive scheduling using synchronization synthesis. *Formal Methods Syst. Des.*, 50(2-3):97–139, 2017.

7 Tej Chajed, M. Frans Kaashoek, Butler W. Lampson, and Nickolai Zeldovich. Verifying concurrent software using movers in CSPEC. In *OSDI*, pages 306–322. USENIX Association, 2018.

8 A. T. Chamillard and Lori A. Clarke. Improving the accuracy of petri net-based analysis of concurrent programs. In *ISSTA*, pages 24–38. ACM, 1996.

9 Qichang Chen, Liqiang Wang, Zijiang Yang, and Scott D. Stoller. HAVE: detecting atomicity violations via integrated dynamic and static analysis. In *FASE*, volume 5503 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2009.

10 Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

11 Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

12 The Coq proof assistant, 2023. URL: `https://coq.inria.fr/`.

13 Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. Views: compositional reasoning for concurrent programs. In *POPL*, pages 287–300. ACM, 2013.

14 Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2009.

15 Tayfun Elmas. QED: a proof system based on reduction and abstraction for the static verification of concurrent software. In *ICSE (2)*, pages 507–508. ACM, 2010.

16 E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *ICALP*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer, 1980.

17 Cormac Flanagan and Stephen N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, pages 256–267. ACM, 2004.

18 Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. *Commun. ACM*, 53(11):93–101, 2010.

19 Cormac Flanagan and Stephen N. Freund. The Anchor verifier for blocking and non-blocking concurrent software. *Proc. ACM Program. Lang.*, 4(OOPSLA):156:1–156:29, 2020.

20 Cormac Flanagan and Stephen N. Freund. Mover logic: A concurrent program logic for reduction and rely-guarantee reasoning (extended version), 2024. `arXiv:2407.08070`.

21 Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. Types for atomicity: Static checking and inference for java. *ACM Trans. Program. Lang. Syst.*, 30(4):20:1–20:53, 2008.

22 Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Exploiting purity for atomicity. In *ISSTA*, pages 221–231. ACM, 2004.

23 Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI*, pages 338–349. ACM, 2003.

24 Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *TLDI*, pages 1–12. ACM, 2003.

25 Stephen N. Freund and Shaz Qadeer. Checking concise specifications for multithreaded software. *J. Object Technol.*, 3(6):81–101, 2004.

26 Patrice Godefroid. Model checking for programming languages using verisoft. In *POPL*, pages 174–186. ACM Press, 1997.

27 Patrice Godefroid and Pierre Wolper. A partial approach to model checking. In *LICS*, pages 406–415. IEEE Computer Society, 1991.

28 Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. In *PLDI*, pages 646–661. ACM, 2018.

29 John Hatcliff, Robby, and Matthew B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 175–190. Springer, 2004.

30 Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. Automated and modular refinement reasoning for concurrent programs. In *CAV (2)*, volume 9207 of *Lecture Notes in Computer Science*, pages 449–465. Springer, 2015.

31 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

32 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and java. In *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.

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

34 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018.

35 Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2006.

36 Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. Inductive sequentialization of asynchronous programs. In *PLDI*, pages 227–242. ACM, 2020.

37 Bernhard Kragl and Shaz Qadeer. Layered concurrent programs. In *CAV (1)*, volume 10981 of *Lecture Notes in Computer Science*, pages 79–102. Springer, 2018.

38 Bernhard Kragl and Shaz Qadeer. The civl verifier. In *FMCAD*, pages 143–152. IEEE, 2021.

39 Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. Synchronizing the asynchronous. In *CONCUR*, volume 118 of *LIPIcs*, pages 21:1–21:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.

40 Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. Refinement for structured concurrent programs. In *CAV (1)*, volume 12224 of *Lecture Notes in Computer Science*, pages 275–298. Springer, 2020.

41 Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

42 Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: low-effort verification of high-performance concurrent programs. In *PLDI*, pages 197–210. ACM, 2020.

43 Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *VMCAI*, volume 9583 of *Lecture Notes in Computer Science*, pages 41–62. Springer, 2016.

44 Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Dependable Software Systems Engineering*, volume 50 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 104–125. IOS Press, 2017.

45 Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, pages 267–280. USENIX Association, 2008.

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

47 Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

48 Doron A. Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390. Springer, 1994.

49 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

50 Amit Sasturkar, Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Automated type-based analysis of data races and atomicity. In *PPoPP*, pages 83–94. ACM, 2005.

51 Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for java-like programs based on dynamic frames. In *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 261–275. Springer, 2008.

52 Viktor Vafeiadis. *Modular fine-grained concurrency verification.* PhD thesis, University of Cambridge, UK, 2008.

53 Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2007.

54 Liqiang Wang and Scott D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *PPoPP*, pages 137–146. ACM, 2006.

55 Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Software Eng.*, 32(2):93–110, 2006.

56 John Wickerson, Mike Dodds, and Matthew J. Parkinson. Explicit stabilisation for modular rely-guarantee reasoning. In *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 610–629. Springer, 2010.

57 James R. Wilcox, Cormac Flanagan, and Stephen N. Freund. Verifiedft: a verified, high-performance precise dynamic race detector. In *PPoPP*, pages 354–367. ACM, 2018.

58 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.

59 Eran Yahav. Verifying safety properties of concurrent java programs using 3-valued logic. In *POPL*, pages 27–40. ACM, 2001.

60 Jaeheon Yi, Tim Disney, Stephen N. Freund, and Cormac Flanagan. Cooperative types for controlling thread interference in java. In *ISSTA*, pages 232–242. ACM, 2012.

61 Jaeheon Yi, Tim Disney, Stephen N. Freund, and Cormac Flanagan. Cooperative types for controlling thread interference in java. *Sci. Comput. Program.*, 112:227–260, 2015.

62 Jaeheon Yi and Cormac Flanagan. Effects for cooperable and serializable threads. In *TLDI*, pages 3–14. ACM, 2010.