Inferring Synchronization Disciplines to Verify Atomicity of Concurrent Code

Margaret Allen '20, David J. Lee '21, Stephen Freund — Williams College

Concurrency

- Running multiple threads concurrently improves performance.
- But if threads share data, they may interfere with each other, leading to nasty bugs.
- Suppose two threads simultaneously attempt to deposit $1 to the same, empty bank account. Their steps may interleave in different ways:

	Interleaving 1 (Good)	Interleaving 2 (Bad)
Thread 1	`x = balance` `//x = 0`	`x = balance` `//x = 0`
Thread 1	`balance = x + 1` `//balance = 1`	`y = balance` `//y = 0`
Thread 2	`y= balance` `//y = 1`	`balance = x + 1` `//balance = 1`
Thread 2	`balance = y + 1` `//balance = 2`	`balance = y + 1` `//balance = 1`

- Interleaving 1 behaves as expected, but Interleaving 2 does not (one of the deposits is lost).
- We want to write **thread-safe** programs: programs that always behave as expected.

Approaches

Mutual-Exclusion Lock

Only one thread may hold the lock at a time. Other threads must wait until the lock is released before they can acquire it.

```
acquire(lock)
tmp = x
x = tmp + 1
release(lock)
```

Optimistic Update (Compare-and-Set)

Store target's value, calculate new value, and update only if the target's value is unchanged.

```
while(true) {
  old = x
  new = old + 1
  success = CAS(x, old, new)
  if(success)
    break
}
```

```
CAS(x, old, new):
  if (old == x) {
    x = new
    success = true
  }
  else {
    success = false
  }
```

Atomicity Guarantees

- We want to ensure that code blocks are free of interference (**atomic**).
- Atomic blocks are **serializable**: their behavior when interleaved with other threads is the same as their behavior with no interleaved steps.

Interleaved		Serialized
`acquire(m)`		`...`
`...`		`acquire(m)`
`tmp = x`	behaves the same as	`tmp = x`
`...`		`x = tmp + 1`
`x = tmp + 1`		`release(m)`
`release(m)`		`...`

- This makes code easier to verify as correct.
- Programmers can identify where block boundaries are by annotating them with `yields`:

```
while(true) {
  old = x
  new = old + 1          } atomic
  yield
  success = CAS(x, old, new)
  if(success) break      } atomic
  yield
}
```

- But how do we know that yields are in the right place?

Verifying Atomicity via Reduction

- We want to verify that each atomic block is indeed serializable.
- To do this, we assign a **commutativity** to every step of the block.
- This captures how a thread's steps can be moved relative to steps in other threads without changing overall behavior.

```
  yield
R acquire(m)
B tmp = x
B x = tmp + 1
L release(m)
  yield
```

Basic Commutativity Rules

- Both-movers (**B**) can commute in either direction (e.g. lock-protected memory access)
- Right-movers (**R**) can commute only forward (e.g. lock acquisition)
- Left-movers (**L**) can commute only backward (e.g. lock release)
- Non-movers (**N**) cannot commute without changing execution behavior (e.g. unprotected variable access)

Reduction

Given code with commutativity annotations, we can use **reduction** to show that a code block is serializable:

Interleaved		Serialized
`[R] acquire(m)`	`...`	`...`
`...`	`[R] acquire(m)`	`[R] acquire(m)`
`[B] tmp = x`	`[B] tmp = x`	`[B] tmp = x`
`...`	`[B] x = tmp + 1`	`[B] x = tmp + 1`
`[B] x = tmp + 1`	`...`	`[L] release(m)`
`[L] release(m)`	`[L] release(m)`	`...`

If a code block matches the pattern `(R|B)*[N](L|B)*`, then it is serializable.
If all atomic blocks in a program are serializable, then the program is **reducible**.

Problem

- The basic commutativity rules do not cover many cases:
  - Read-only data: reads are both-movers but writes are illegal accesses.
  - Thread-local data: within the thread, reads and writes are both-movers. In other threads, reads and writes are illegal accesses.
- To use reduction for these cases, we need more sophisticated specifications of commutativity behaviors.
- Such specifications are difficult for programmers to write.

Objective

Develop an algorithm to automatically infer a **synchronization discipline** for a concurrent program, enabling reduction-based verification of its atomicity requirements.

Applications

- Verify thread safety of arbitrary code.
- Make implicit synchronization disciplines explicit, making code easier to read and maintain.

Synchronization Disciplines

- A synchronization discipline is a policy defining the commutativities of accesses to shared data, given the context in which the access occurs.
- We want to infer a discipline for each shared variable.
- We extend basic commutativity rules with conditional rules capturing context information.

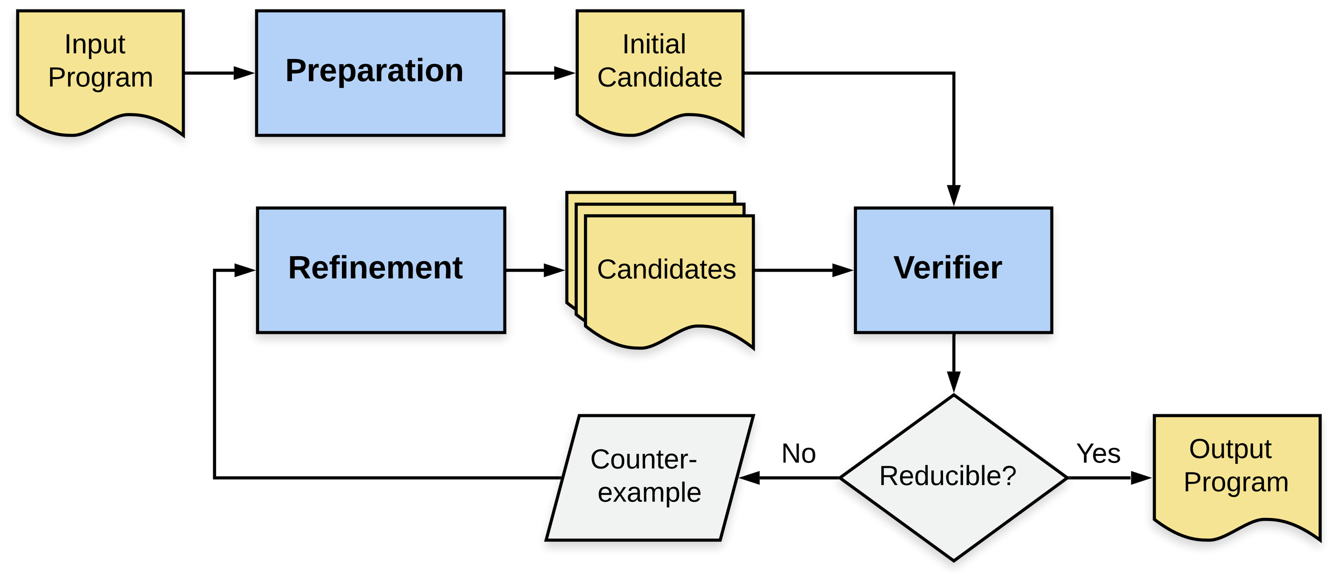
Examples

Note: E stands for "error", i.e., an illegal memory access.

`x => holds(m) ? B : E`	x is protected by lock m
`x => isLocal() ? B : (isRead() ? B : E)`	B if thread-local, otherwise read-only
`x => holds(m) ? (isRead() ? B : N) : (isRead() ? N : E)`	x is write-protected by lock m
`x => (x == -1) ? (newValue == -1 ? R : E) : N`	x cannot change once set to -1

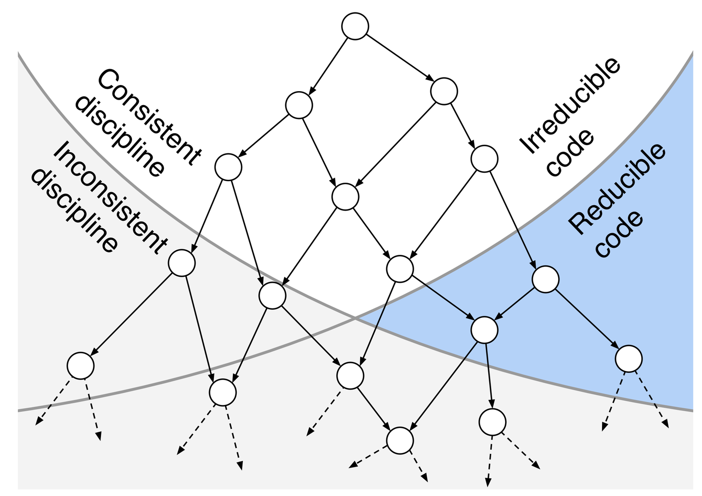
Algorithm

- Create an initial candidate using basic predicates such as `isRead` and `isLocal`, along with predicates mined from the input program.
- Use **counterexample-guided inductive synthesis** to generate better candidates using verifier feedback on earlier candidates.



Counterexample-Guided Inductive Synthesis

- A discipline is **consistent** if accesses do in fact commute as specified without interference.
- Start with a discipline where no accesses are allowed. (This discipline is consistent but irreducible.)
- Refine to allow accesses based on verification errors.
- Perform a directed search of possible disciplines to find one that is consistent and reducible.



Initial Candidate

```
x => holds(m) ?
       isRead? E : E
     : isRead? E : E

[R] acquire(m)     (1)
[E] tmp = x        (2)
[E] x = tmp + 1    (3)
[L] release(m)     (4)
    yield          (5)
[E] tmp = x        (6)
```

→ **Verifier** → Reduction Error on lines 1-4

1st-Gen Candidates

```
x => holds(m) ?
       isRead? N : E
     : isRead? E : E
...
```

...

```
x => holds(m) ?
       isRead? E : N
     : isRead? E : E
...
```

...

nth-Gen Candidates

```
x => holds(m) ?
       isRead? B : N
     : isRead? N : E
...
```

→ **Verifier** → Consistency Error

```
tmp = x            (1)
[B] x = tmp + 1    (2)
```

Commuting (2) before (1) changes the value assigned to tmp in (1), so (2) cannot be a both-mover.

...

```
x => holds(m) ?
       isRead? N : B
     : isRead? N : E
...
```

→ **Verifier** → Success!

- Our system performs well on small- to medium-sized programs. We are working towards reducing the search space for larger programs.

This work was supported by NSF grant 1812951. Some examples adapted from Nyhus '20.