Modular Verification with Abstract Interference Models

Computer Science and Engineering ■ College of Engineering ■ The Ohio State University

Case Study: A Concurrent Queue

Alan Weide*, Murali Sitaraman†, *Paul Sivilotti**

*Ohio State University

†Clemson University

Acknowledgement: U.S. NSF grant CCF-1161916

Context: Perfect Parallelism

- Concurrent threads do not modify shared state

```
split(problem, p1, p2);
cobegin {
  solve(p1, s1);
  solve(p2, s2);
}
merge(s1, s2, solution);
```

- Benefits
  - Minimal synchronization
    - No locks, semaphores, mutexes, signals, *etc*
    - No deadlocks, fewer synchronization bottlenecks
  - Determinism
    - Same semantics as sequential execution
    - No race conditions, easier to debug

Challenge: Aliasing

- Client code:

```
cobegin {
  update(p1);
  update(p2);
}
```

- Threads are independent only if p1 and p2 are fully distinct

Challenge: Deliberate Sharing

- In client code:

```
split(list, list1, list2);
cobegin {
  countIn(list1, item, c1);
  countIn(list2, item, c2);
}
occurrences = c1 + c2;
```

- Spec of `countIn`

```
countIn(list: List(T), i: T, c: Integer)
ensures list = #list and
        i = #i and
        c = count(i, list)
```

- Correctness of parallel composition depends on implementation of `countIn`
  - Must *preserve* `i`, not just *restore* it

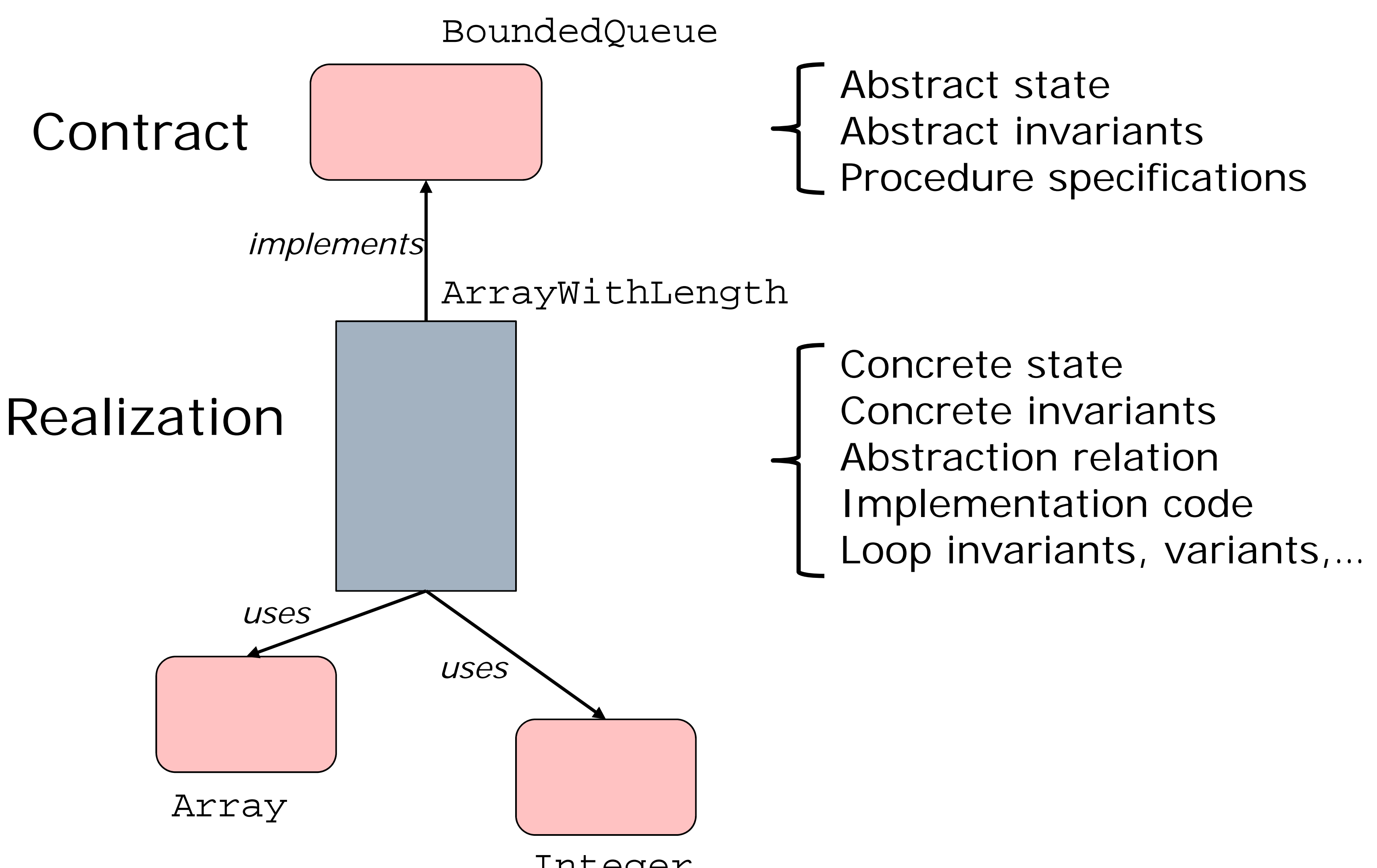
Related Work

- Type systems
  - Permissions, DPJ, Liquid Effects
- Deterministic scheduling
  - DMP, CoreDet, Kendo
  - Grace, Determinator, DOMP
- OS/Scheduling
  - Dthreads, dOS, Legion
- Non-blocking data structures
  - CAS, linearizability, transactional memory
- Assertional
  - Non-interference VCs, Bridge assertions
- Separation logic
  - CSL, Concurrent abstract predicates
- Value semantics
  - ParaSail, dataflow and futures

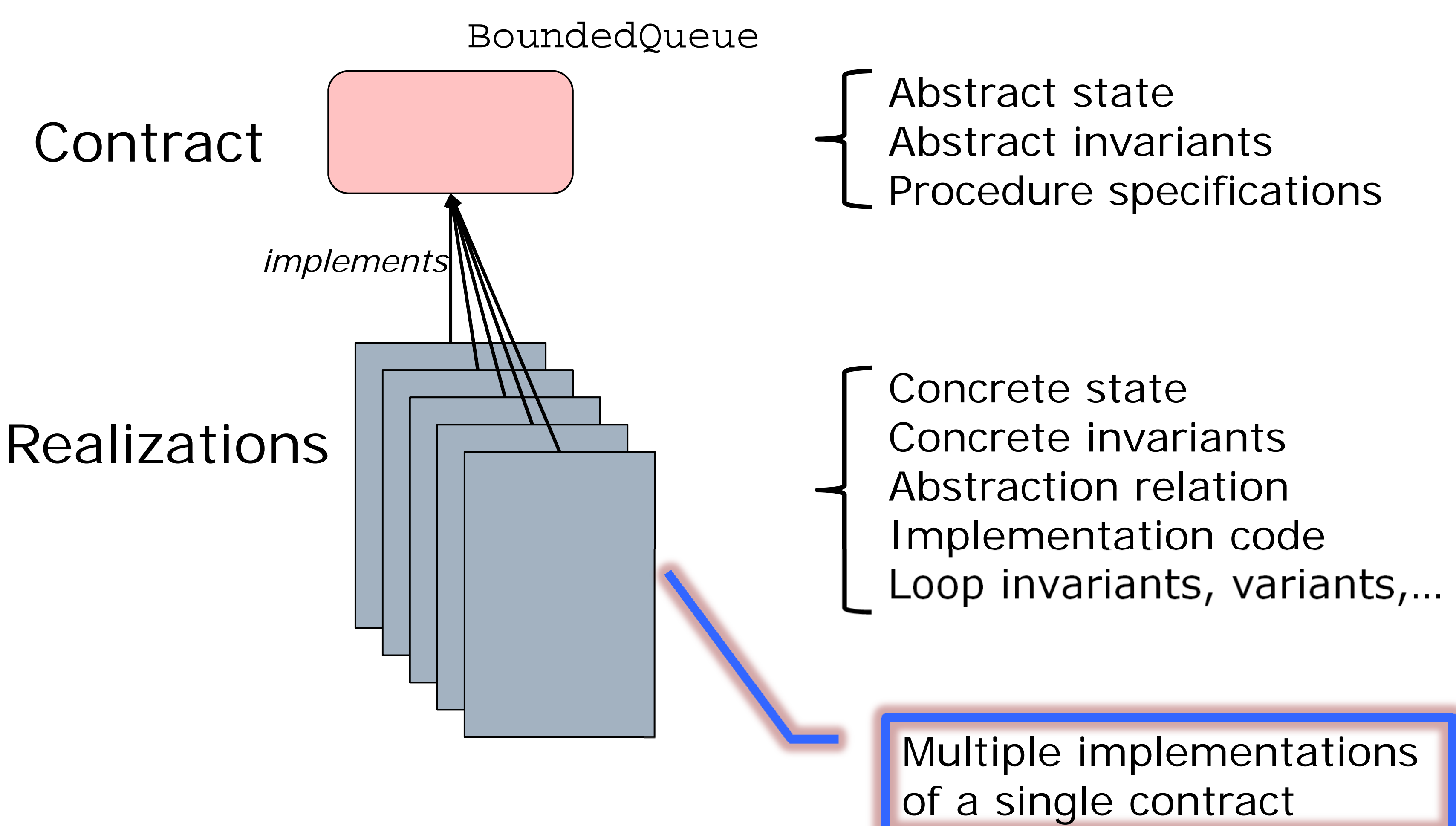
Approach: Extend RESOLVE

- Value semantics
  - No aliases
- Constant-time swap primitive
- Modularity
  - Program types modelled by mathematical abstractions
  - Proofs of implementations layered on math models of other concepts
- Tool support: Static verification workbench
  - https://www.cs.clemson.edu/resolve
  - http://resolveonline.cse.ohio-state.edu

Modularity



Modularity



Exposing Entanglement

Intermediate Model

detangles

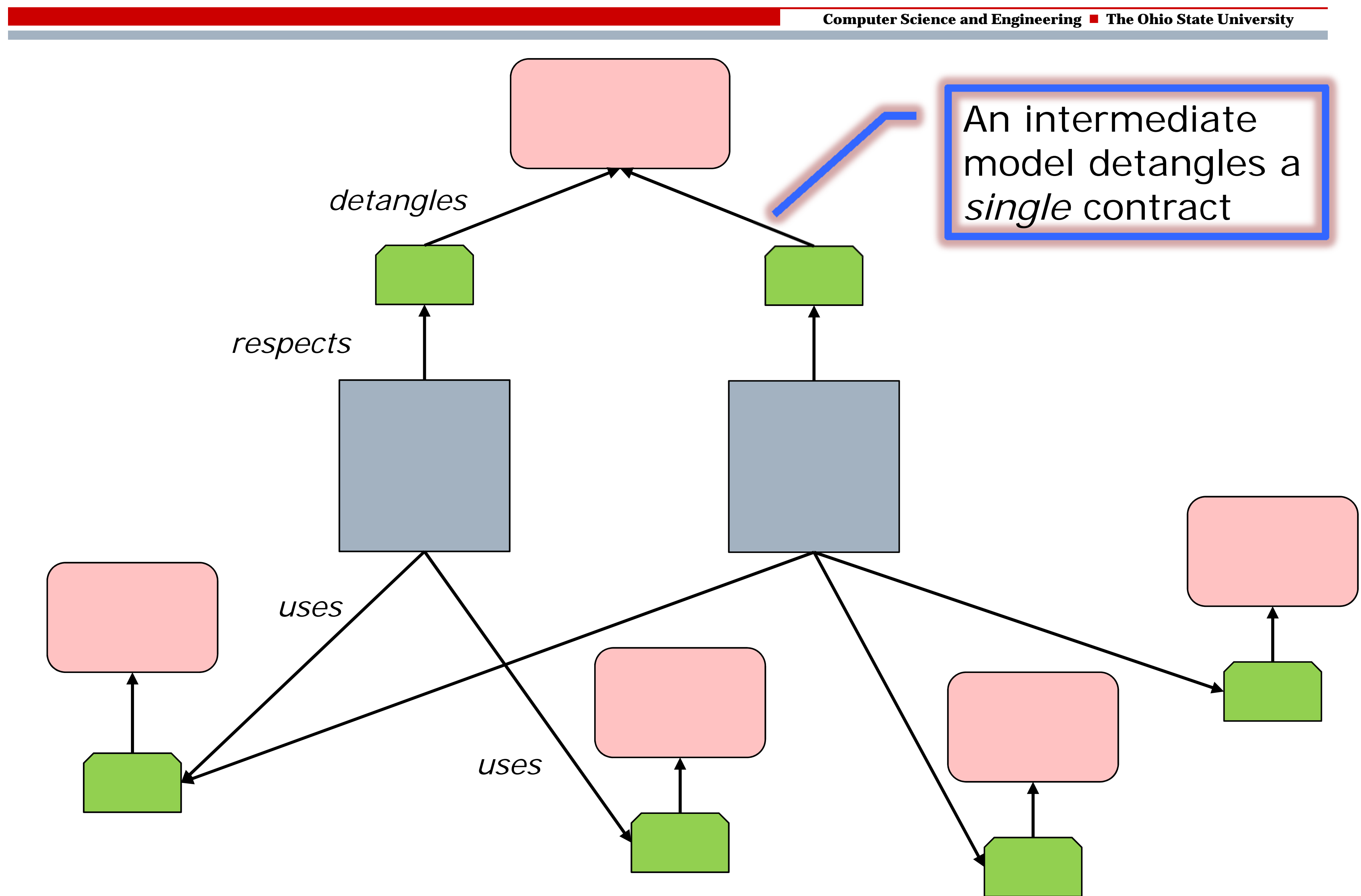
Partitions
Operation effects

respects

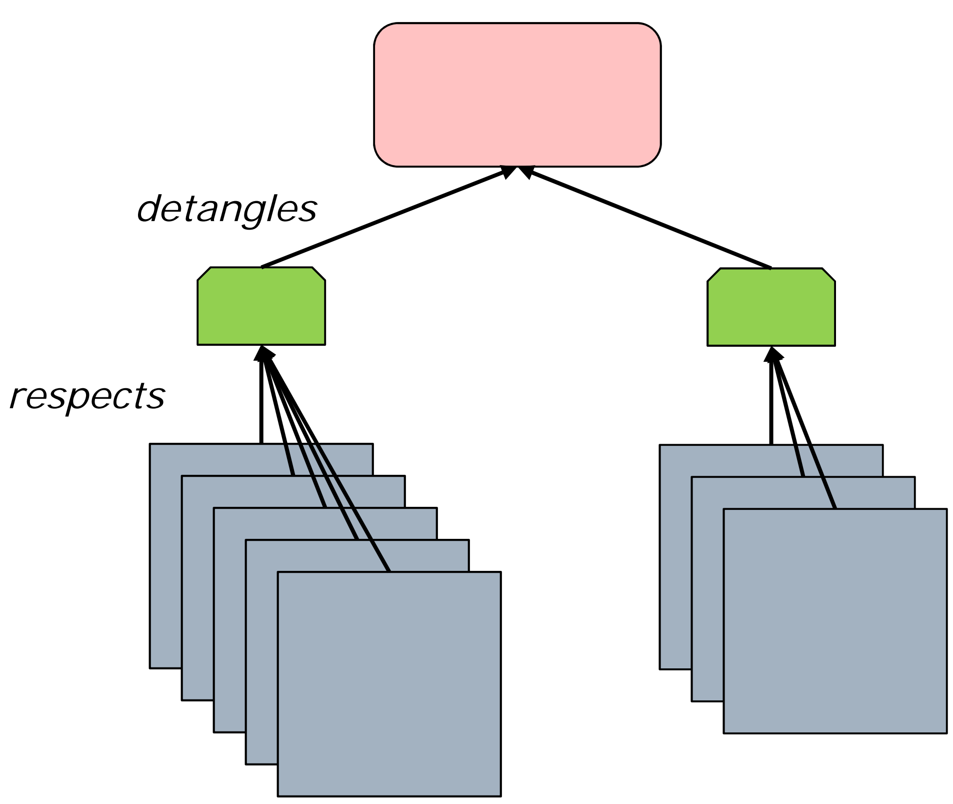
uses

uses

Exposing Entanglement



Exposing Entanglement



Illustrative Example: A Bounded Concurrent Queue


```
type Queue is modeled by string of Item
  exemplar q
  constraint |q| <= MAX_LENGTH
  initially q = empty_string
```

⟨ , , , , ⟩

```
procedure Enqueue (clears e: Item,
                   updates q: Queue)
  requires |q| < MAX_LENGTH
  ensures q = #q * <#e>
```

$\#e$

$\#q$ ⟨ , , ⟩

e

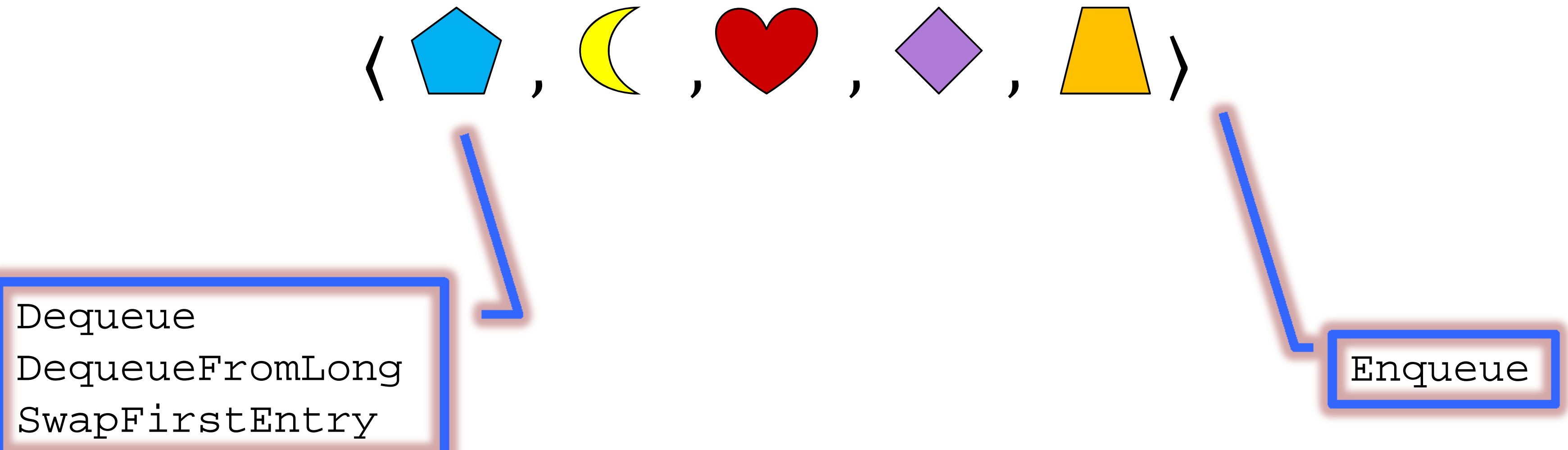
q ⟨ , , , ⟩

Other Bounded Queue Operations


```
procedure Dequeue (replaces r: Item,
                   updates q: Queue)
  requires q /= empty_string
  ensures #q = <r> * q

procedure DequeueFromLong (replaces r: Item,
                           updates q: Queue)
  requires |q| >= 2
  ensures #q = <r> * q

procedure SwapFirstEntry (updates e: Item,
                          updates q: Queue)
  requires q /= empty_string
  ensures <e> = #q[0, 1) and
          q = <#e> * #q[1, |#q|)
```

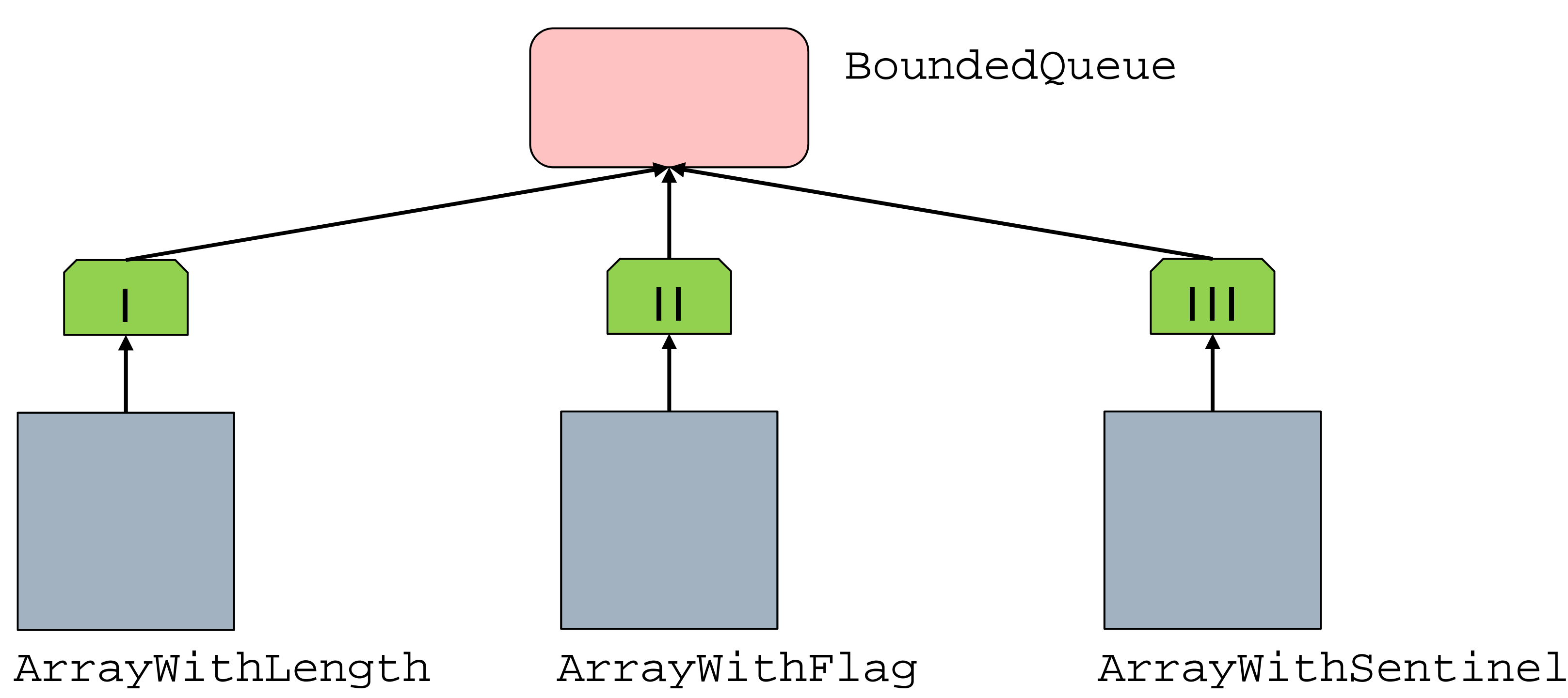
Decoupled Head and Tail

- Consider a non-empty queue



- Which operations are perfectly parallel with `Enqueue`?

Different Partitions



Different Partitions

BoundedQueue

I

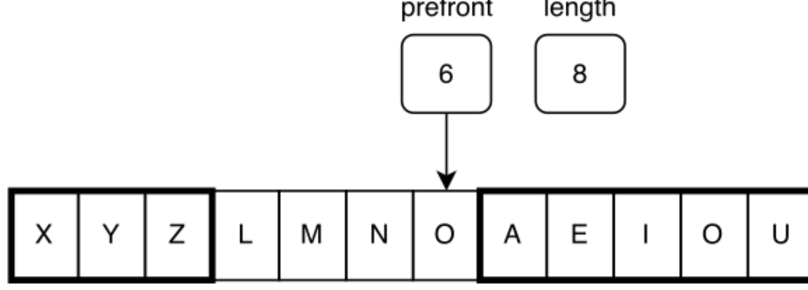
II

III

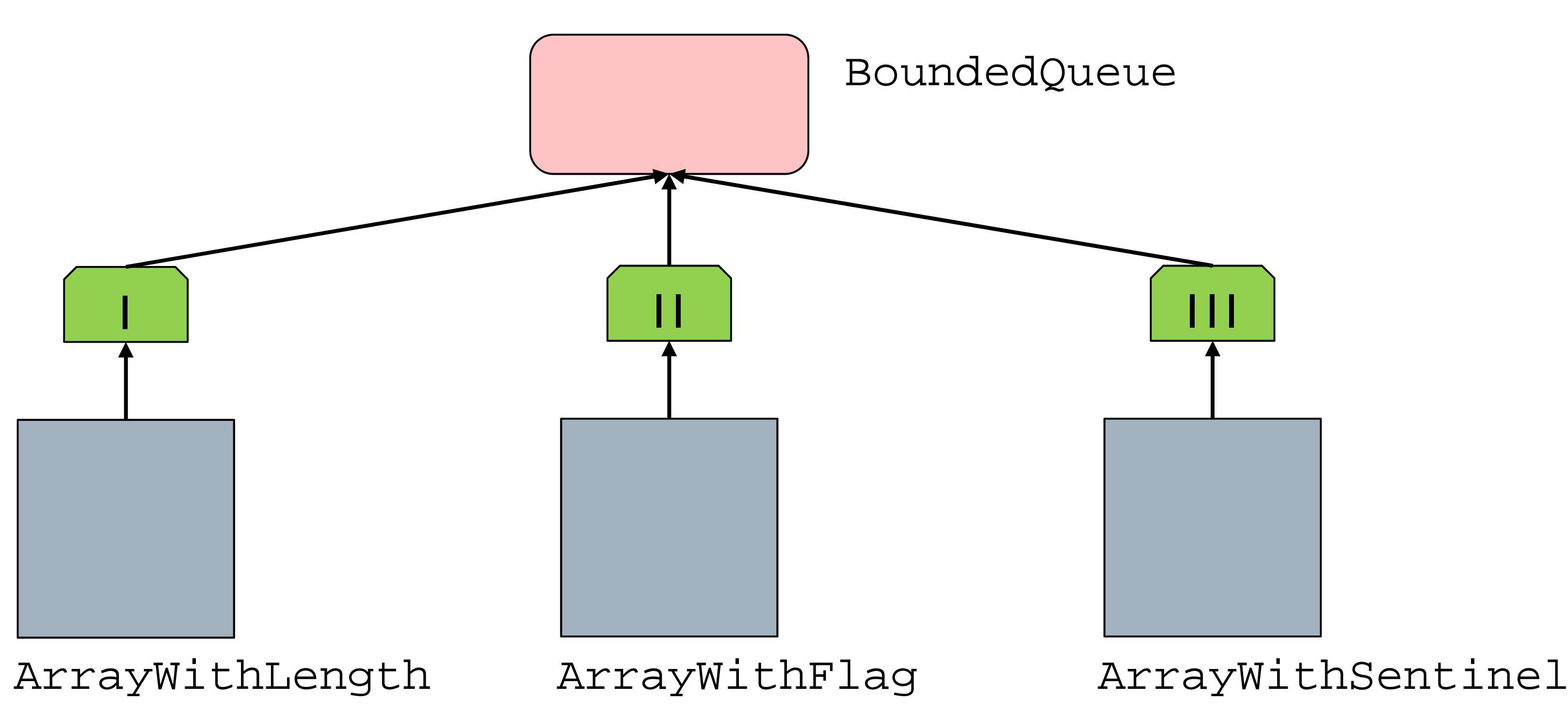
ArrayWithLength

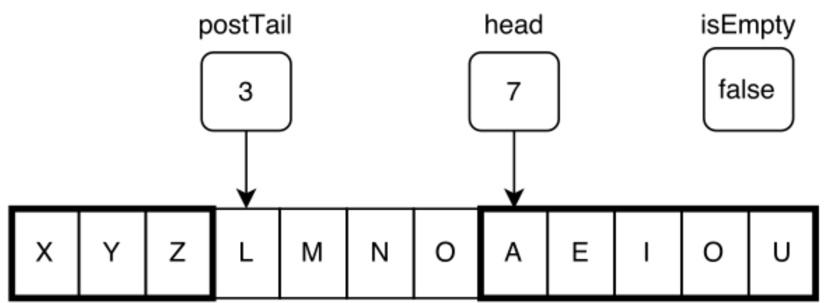
ArrayWithFlag

ArrayWithSentinel

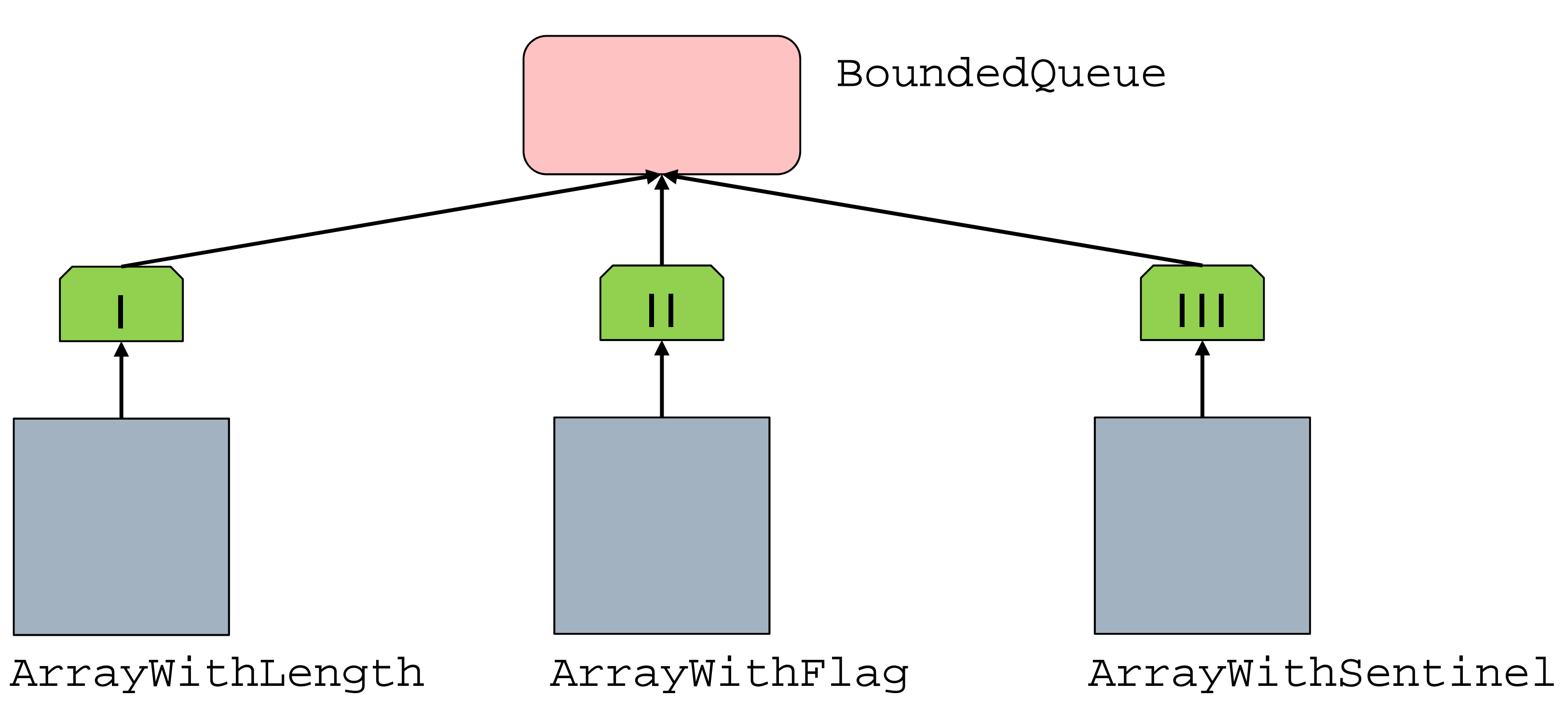


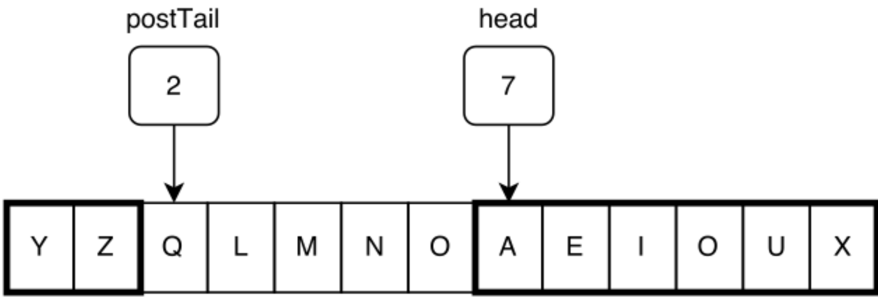
Different Partitions





Different Partitions





Perfectly Parallel with `Enqueue`

	I	II	III
`Dequeue`			✓
`DequeueFromLong`		✓	✓
`SwapFirstEntry`	✓	✓	✓
	…Length	…Flag	…Sentinel

Partition I


```
partition for Queue is (head, tail, index)



procedure Enqueue (clears e: Item,
                   updates q: Queue)
  affects q.tail
  preserves q.index
  when |q| = 0 affects q.head



procedure Dequeue (replaces r: Item, updates q: Queue)
  affects q.head, q.tail, q.index




procedure DequeueFromLong (replaces r: Item, updates q: Queue)
  affects q.head, q.tail, q.index

procedure SwapFirstEntry (updates e: Item, updates q: Queue)
  affects q.head
  preserves q.index
```

Partition I: Conditional Effects

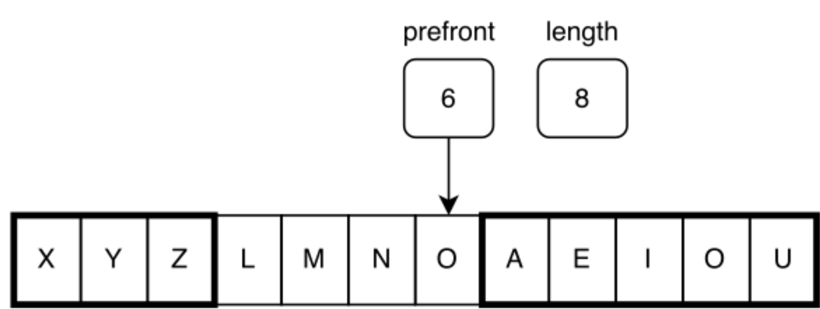

```
partition for Queue is (head, tail, index)



procedure Enqueue (clears e: Item,
                   updates q: Queue)
  affects q.tail
  preserves q.index
  when |q| = 0 affects q.head



procedure Dequeue (replaces r: Item, updates q: Queue)
  affects q.head, q.tail, q.index




procedure DequeueFromLong (replaces r: Item, updates q: Queue)
  affects q.head, q.tail, q.index

procedure SwapFirstEntry (updates e: Item, updates q: Queue)
  affects q.head
  preserves q.index
```

Partition I: Non-Interference


```
partition for Queue is (head, tail, index)



procedure Enqueue (clears e: Item,
                   updates q: Queue)
  affects q.tail
  preserves q.index
  when |q| = 0 affects q.head



procedure Dequeue (replaces r: Item, updates q: Queue)
  affects q.head, q.tail, q.index



procedure DequeueFromLong (replaces r: Item, updates q: Queue)
  affects q.head, q.tail, q.index

procedure SwapFirstEntry (updates e: Item, updates q: Queue)
  affects q.head
  preserves q.index
```

ArrayWithLength Realization


```
partition for Queue is (head, tail, index)
```




```
procedure Enqueue (clears e: Item,
                   updates q: Queue)
  affects q.tail
  preserves q.index
  when |q| = 0 affects q.head
```

```
procedure Dequeue (replaces r: Item, updates q: Queue)
  affects q.head, q.tail, q.index
```

```
procedure DequeueFromLong (replaces r: Item, updates q: Queue)
  affects q.head, q.tail, q.index

procedure SwapFirstEntry (updates e: Item, updates q: Queue)
  affects q.head
  preserves q.index
```

ArrayWithLength Realization

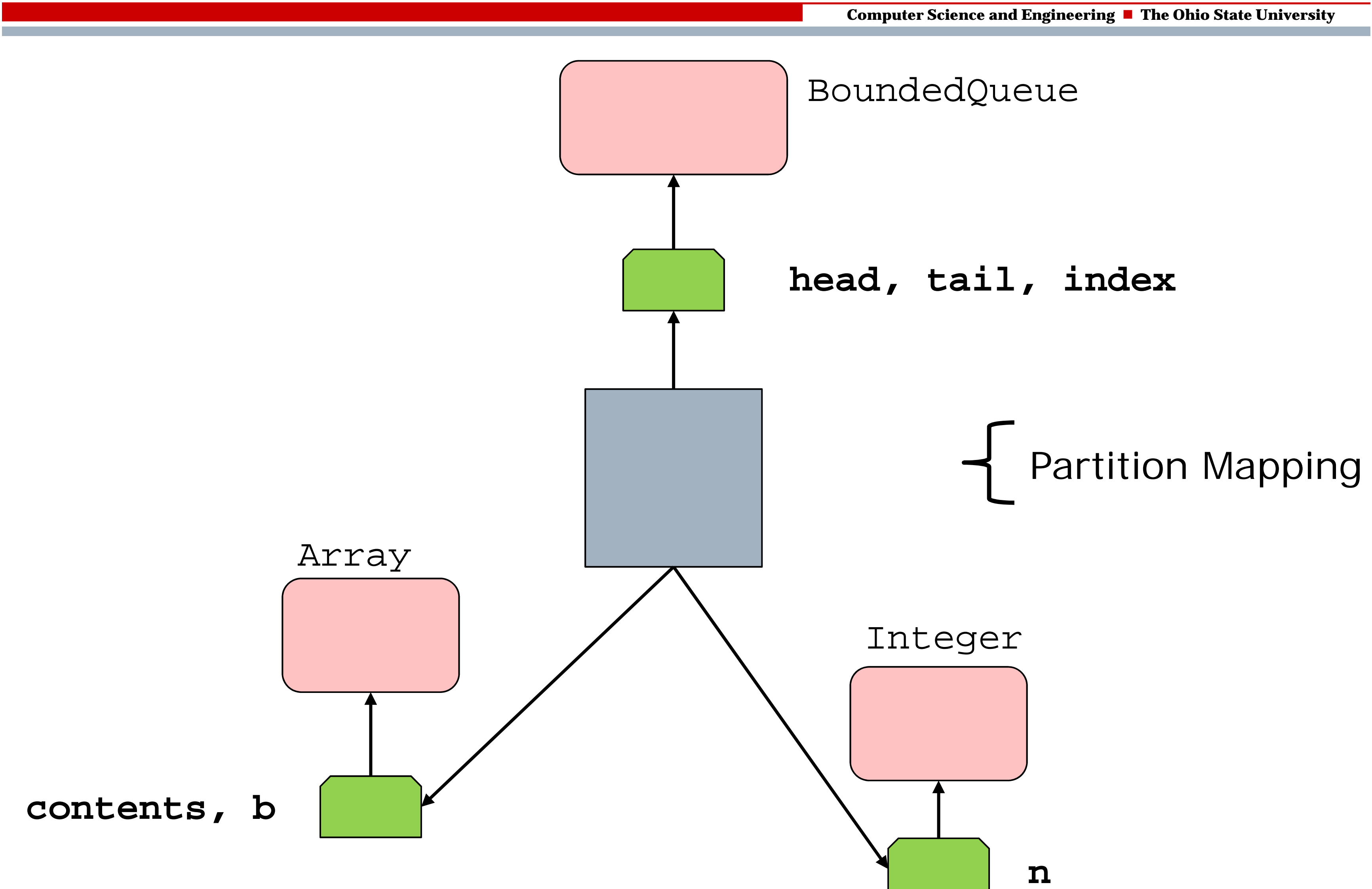

```
partition for Queue is (head, tail, index)
```

prefront 6 length 8

X Y Z L M N O A E I O U

```
procedure Enqueue (clears e: Item,
                   updates q: Queue)
  affects q.tail
  preserves q.index
  when |q| = 0 affects q.head

procedure Dequeue (replaces r: Item, updates q: Queue)
  affects q.head, q.tail, q.index

procedure DequeueFromLong (replaces r: Item, updates q: Queue)
  affects q.head, q.tail, q.index

procedure SwapFirstEntry (updates e: Item, updates q: Queue)
  affects q.head
  preserves q.index
```

Layering of Intermediate Models



Partition II


```
partition for Queue is (head, tail, m)

procedure Enqueue (clears e: Item,
                   updates q: Queue)
  affects q.tail;
  when |q| = 0
    affects q.head, q.m
  otherwise
    preserves q.m

procedure Dequeue (replaces r: Item, updates q: Queue)
  affects q.head;
  when |q| = 1
    affects q.tail, q.m
  otherwise
    preserves q.tail

procedure DequeueFromLong (replaces r: Item, updates q: Queue)
  affects q.head

procedure SwapFirstEntry (updates e: Item, updates q: Queue)
  affects q.head
```

Partition III


```
partition for Queue is (head, tail)
```

```
procedure Enqueue (clears e: Item,
                   updates q: Queue)
  affects q.tail;
  when |q| = 0
    affects q.head
```

postTail
2
head
7

Y	Z	Q	L	M	N	O	A	E	I	O	U	X

```
procedure Dequeue (replaces r: Item, updates q: Queue)
  affects q.head
```

```
procedure DequeueFromLong (replaces r: Item, updates q: Queue)
  affects q.head

procedure SwapFirstEntry (updates e: Item, updates q: Queue)
  affects q.head
```

Summary

- Leveraging RESOLVE framework
  - Verification system for sequential correctness, client code and implementation
  - See demo this afternoon
- Work in progress
  - Inclusion of *segmented* fields in partition descriptions
  - Development of modular proof system and generation of VCs
    - Proof of perfect parallelism in client code using partition descriptions
    - Proof that implementation respects partition descriptions
  - Integration with existing RESOLVE IDE