# Array Abstractions to Simplify Reasoning About Concurrent Client Code

Alan W. Weide, Paolo A. G. Sivilotti, and Murali Sitaraman

**Technical Report RSRG-17-04**
School of Computing
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

December 2017

# Array Abstractions to Simplify Reasoning About Concurrent Client Code

Alan Weide[1], Paolo A. G. Sivilotti[1], and Murali Sitaraman[2]

[1] The Ohio State University, Columbus OH 43221, USA
weide.3@osu.edu, paolo@cse.ohio-state.edu
[2] Clemson University, Clemson SC 29634, USA
murali@clemson.edu

**Abstract.** The overall research objective of the RESOLVE project is to enable automated verification of sequential and parallel object-based programs in a modular fashion. While we can leverage *clean semantics* to dramatically simplify the verification of sequential programs by eliminating aliases, shared state is unavoidable in a concurrent context and must be dealt with in a modular, abstract way. To this end, the RESOLVE project has introduced *interference contracts* which define a partitioning of the representation of a data type into disjoint partitions and the effects that the execution of an operation might have on these partitions. Interference contracts have proved useful in exposing the maximum available parallelism for a variety of components, but some cases require a much more flexible framework. As a first step toward a fully general interference contract, several new array abstractions have been developed which simplify reasoning about concurrent code and which require as few new capabilities in interference contracts as possible. Further, a `BoundedQueue` implementation was built on top of one of these abstractions in such a way that it achieves as much avaialble synchronization-free parallelism as possible.

## 1 Introduction

Traditional parallel implementations of certain kinds of algorithms (e.g., divide-and-conquer, map-reduce, and producer-consumer) are often difficult for automated verification systems to reason about because of the subtleties involved in determining which parts of the data are actually accessed through a call to the method. In an effort to simplify such reasoning, we introduce several novel array abstractions which give the client the ability to partition the array into arbitrary parts based on some client-defined indices, and to operate on those two arrays in parallel and without fear of interference.

### 1.1 Clean semantics

One of the core principles of the RESOLVE programming language is the notion of *clean semantics*. When reasoning with clean semantics, a programmer (or

formal verification engine) can rely on the fact that two different variables may be treated as two independent entities. The verification of the correctness of the parallel programs in this paper relies heavily on this idea and leverages it to dramatically simplify reasoning and maintain modularity.

Although data sharing can normally be avoided in sequential programs, in parallel programs the sharing of data among threads is sometimes necessary to maximize performance. Thus, showing determinism (thereby enabling verification) might require exposing some implementation details to the client about how the data is split up. It is ciritical for tractability that the exposing of these details be modular; that is, the details revealed by one realization should be specified only in terms of the details exposed by components it is layered upon. Abstraction must also be preserved so as not to complicate reasoning about the independence of concurrent operations.

### 1.2 The Interference Contract

The *interference contract* specification construct divides the representation space of a data type into a number of *partitions*, each of which is disjoint from the others in the sense that at the representation level, each piece of data (e.g., each representation field[3]) is a member of exactly one partition. An interference contract extends abstract specifications to include effects summaries, which define how an operation will interact with the partitions of an object. These summaries are described in terms of three partition modes: *affects*, *preserves*, and *oblivious to*. In addition to these three modes, a partition is said to have a *restructures* effect if, by executing the operation, the members of that partition might be moved to a different one, or that partition might have another partition's members moved into it, though none of the *values* of those members will have changed. All three partition modes and the *restructures* effect may be conditional on the abstract values of the parameters.

Interference contracts find their primary use in `cobegin` blocks. The rules associated with `cobegin` blocks in RESOLVE code using interference contracts are relatively straightforward. Each partition of objects mentioned in a cobegin block's constituent statements may be *affects*-mode in at most one statement, and if it is *affects*-mode in some statement then it must be *oblivious*-mode in all others. When these properties are satisified, the statements in the cobegin block are said to be *non-interfering* and their execution will be deterministic. Once non-interference (and therefore determinism) is established using the interference contract, verification can proceed as if the execution of the operations were sequential.

**The Restructures Effect** In component implementations that make use of dynamically assigned partitions, an interesting situation can occur: a method might change a partition's membership without altering the values of any fields that

[3] Notice that because interference contracts are modular, representation fields might themselves have partitions that may be members of a partition "one level up".

live in this partition. In this case, we say that partition is under the *restructures* effect.

The proof rules concerning the resturctures effect are subtle. In some cases, it may not be apparent that there are any issues that arise from changing the membership of a partition. Consider a reasonable realization of `Splittable_Array` (specified in Listing 1; its interference contract is in Listing 2). The realization probably uses an underlying array and place the entries at indices less than `Split_Point` in the `Low` partition and the rest of the indices in the `Up` partition. These partitions, then, are dynamically assigned (even though they may not appear to be so at first glance) and there may be an operation that changes partition membership. This operation is `Set_Split_Point`. By changing the split point of the array, the members of the `Low` and `High` partitions are necessarily changed, however the operation does *not* change the values of the entries themselves. If this change occurs as the last line of the method body, reasoning is still sound when it relies on membership not changing through the course of the method. However, when this method body is used in a layered implementation which itself is intended for use in a parallel context, it's possible that this layered method body will later attempt to modify some entry of the array. But becuase `Set_Split_Point` changes the membership of the partitions, there is no guarantee that this entry is still in the partition it was in before the call to `Set_Split_Point`. Thus, if a partition has the *restructures* effect in some statement in a method body, all subsequent statements in that method body must have it as an *oblivious*-mode partition.

## 2 Array Abstractions

This report introduces three different array abstractions, each of which can simplify reasoning about a class of parallel programs.

### 2.1 Splittable Array

The first abstraction is the `Splittable_Array` abstraction, which can simplify reasoning about recursive and concurrent divide-and-conquer algorithms. The `SplittableArray` component provides the client with operations which can be used to divide the array at some split point into two sub-arrays with contiguous indices, which by virtue of RESOLVE's *clean semantics* may be reasoned about as totally independent objects, even though in a reasonable implementation they would be two parts of the same underlying array.

Two parts of this component are presented here: the abstract specification (the "`Concept`") and the `interference contract`. A general, in-depth explanation of the intereference contract is available in [1].

**Listing 1.** Abstract specification for `Splittable_Array`

```
Concept Splittable_Array_Template(type Entry);
```

```
var Ids: powerset(Z);
  initialization ensures Ids = empty_set;

Type Family Splittable_Array is modeled by Cart_Prod
  Id: Z;
  Lower_Bound, Upper_Bound: Z;
  Contents: Z -> Entry;
  Split_Point: Z;
  Lower_Part_In_Use: B;
  Upper_Part_In_Use: B;
end;
  exemplar A;
    constraint A.Lower_Bound <= A.Upper_Bound
      and A.Lower_Bound <= A.Split_Point
      and A.Split_Point <= A.Upper_Bound
      and A.Id in Ids;
    initialization ensures
      A.Lower_Bound = 0 and A.Upper_Bound = 0
      and not A.Lower_Part_In_Use
      and not A.Upper_Part_In_Use
      and A.Id not in #Ids;
end;

Operation Set_Bounds(evaluates LB, UB: Integer, updates A:
  Splittable_Array);
  requires LB <= UB and not A.Lower_Part_In_Use and not A.
  Upper_Part_In_Use;
  ensures A.Lower_Bound = LB and A.Upper_Bound = UB and
    not A.Lower_Part_In_Use and not A.Upper_Part_In_Use
    not A.Id in #Ids and not #A.Id in Ids;

Operation Set_Split_Point(evaluates i: Integer, updates A:
  Splittable_Array);
  requires A.Lower_Bound <= i and i <= A.Upper_Bound and
    not A.Lower_Part_In_Use and not A.Upper_Part_In_Use;
  ensures A.Split_Point = i and
    [[everything else about A stays the same]];

Operation Swap_Entry_At(evaluates i: Integer, updates A:
  Splittable_Array, updates E: Entry);
  requires not A.Lower_Part_In_Use and not A.Upper_Part_In_Use
  and
    A.Lower_Bound <= i and i < A.Upper_Bound;
  ensures E = #A.Contents(i) and A.Contents(i) = #E and
    [[everything else about A stays the same]];
```

```
Operation Split(updates A: Splittable_Array, replaces L, U:
  Splittable_Array);
  requires not A.Parts_In_Use;
  ensures A.Parts_In_Use
    L.Incl_Lower_Bound = A.Incl_Lower_Bound and
    L.Excl_Upper_Bound = A.Split_Point and
    U.Lower_Bount = A.Split_Point and
    U.Upper_Bount = A.Excl_Upper_Bound and
    L.Id = A.Id and L.Contents = A.Contents and
    U.Id = A.Id and U.Contents = A.Contents
    not L.Parts_In_Use and not U.Parts_In_Use
    [[everything else about A stays the same]];

Operation Combine(updates A: Splittable_Array, clears L, U:
  Splittable_Array);
  requires A.Parts_In_Use and
    not L.Parts_In_Use and not U.Parts_In_Use and
    L.Incl_Lower_Bound = A.Incl_Lower_Bound and
    L.Excl_Upper_Bound = A.Split_Point and
    U.Incl_Lower_Bound = A.Split_Point and
    U.Excl_Upper_Bound = A.Excl_Upper_Bound and
    L.Id = A.Id and U.Id = A.Id;
  ensures not A.Parts_In_Use and
    A.Contents = lambda(i: Z) (if i < A.Split_Point
      then #L.Contents(i) else #U.Contents(i)) and
    [[everything else about A stays the same]];

Operation Lower_Bound(preserves A: Splittable_Array): Integer;
  ensures Lower_Bound = A.Lower_Bound;

Operation Upper_Bound(preserves A: Splittable_Array): Integer;
  ensures Upper_Bound = A.Upper_Bound;

Operation Split_Point(preserves A: Splittable_Array): Integer;
  ensures Split_Point = A.Split_Point;

Operation Lower_Part_Is_In_Use(preserves A: Splittable_Array):
  Boolean;
  ensures Lower_Part_Is_In_Use = A.Lower_Part_In_Use;

Operation Upper_Part_Is_In_Use(preserves A: Splittable_Array):
  Boolean;
  ensures Upper_Part_Is_In_Use = A.Upper_Part_In_Use;
```

```
  Operation Ids_Match(preserves A1, A2: Splittable_Array): Boolean
    ;
    ensures Ids_Match = (A1.Id = A2.Id);

end Splittable_Array_Template;
```

**Listing 2.** An interference contract for `Splittable_Array_Template` which exposes the expected parallelism.

```
Interference Contract Upper_Lower_Separate
  Detangles Splittable_Array_Template;

  Partition for Splittable_Array is {Low, Up, Aux};

  Operation Set_Bounds(evaluates LB, UB: Integer, updates A:
    Splittable_Array);
    affects A@Low, A@Up, A@Aux;

  Operation Set_Split_Point(evaluates p: Integer, updates A:
    Splittable_Array);
    affects A@Aux;
    restructures A@Low, A@Up;

  Operation Swap_Entry_At(evaluates i: Integer, updates A:
    Splittable_Array, updates E: Entry);
    when i < A.Split_Point:
      affects A@Low;
    otherwise:
      affects A@Up;

  Operation Split(updates A: Splittable_Array, replaces L, U:
    Splittable_Array);
    affects A@Low, A@Up;
    preserves A@Aux;

  Operation Combine(updates A: Splittable_Array, clears L, U:
    Splittable_Array);
    affects A@Low, A@Up;
    preserves A@Aux;

  Operation Lower_Bound(preserves A: Splittable_Array): Integer;
    preserves A@Aux;

  Operation Upper_Bound(preserves A: Splittable_Array): Integer;
    preserves A@Aux;
```

```
  Operation Split_Point(preserves A: Splittable_Array): Integer;
    preserves A@Aux;

  Operation Parts_Are_In_Use(preserves A: Splittable_Array):
    Boolean;
    preserves A@Low, A@Up, A@Aux;

  Operation Ids_Match(preserves A1, A2: Splittable_Array): Boolean
    ;
    preserves A1@Aux, A2@Aux;

end Upper_Lower_Separate;
```

Most of this interference contract is straightforward. However, the interference specification for `Swap_Entry_At` is interesting for several reasons. First, it is conditional on the values of `i` and `A`, which necessitates reasoning about the abstract values of objects when verifying parallel code which uses that operation. Second, the body of a reasonable implementation of the operation based on a more traditional array component wouldn't exhibit conditional behavior—it would simply swap the entry at the appropriate position.

Using `Splittable_Array`, it is possible to implement a recursive, parallel divide-and-conquer algorithm to clear all occurences of a given entry in an Array, and to verify that it is a correct implementation of such an algorithm. An important part in the verification of this code's correctness is showing that the operations inside the **cobegin** block are *non-interfering* as defined in [1]. Notice that by the **interference spec** of `Clear_Occurences`, we know that it *affects* all partitions of `A` and *preserves* all partitions of `e`. Thus, even though `e` is an argument in both calls to `Clear_Occurences`, that is not a problem. Further, the two calls to `Clear_Occurences` inside the **cobegin** statement operate on different array variables, so they are necessarily independent because of RESOLVE's *clean semantics*.

**Listing 3.** A recursive, parallel divide-and-conquer solution using `Splittable_Array`

```
Clear_Occurs solution
  Uses Entry with [[some interference contract]];
  Uses Splittable_Array with Upper_Lower_Separate;

  Operation Clear_Occurences(preserves e: Entry; updates A:
    Splittable_Array);
    requires A.Incl_Lower_Bound < A.Excl_Upper_Bound and
      not A.Lower_Part_In_Use and not A.Upper_Part_In_Use;
    ensures [[e does not appear in A]] and
      [[everywhere e appeared in #A is cleared in A]];
    interference spec
      affects A@Low, A@Up, A@Aux;
      preserves e@*;
```

```
  Recursive Procedure
    decreasing A.Excl_Upper_Bound − A.Incl_Lower_Bound;

    if (Excl_Upper_Bound(A) − Incl_Lower_Bound(A) > 1) then
      var A1, A2: Splittable_Array;
      mid := (Incl_Lower_Bound(A) + Excl_Upper_Bound(A)) / 2;
      Set_Split_Point(mid, A);
      Split(A, A1, A2);
      cobegin
        Clear_Occurences(e, A1);
        Clear_Occurences(e, A2);
      end;
      Combine(A, A1, A2);
    else
      var LB: Integer;
      LB := Incl_Lower_Bound(A);
      var ae: Entry;
      Swap_Entry_At(A, LB, ae);
      If Are_Equal(e, ae) then
        Clear(ae);
      end;
      Swap_Entry_At(A, LB, ae);
    end;
  end Occurrences;
end Occurs_Ct;
```

### 2.2 Distinguished Index Array

Next, we introduce the `Distinguished_Index_Array` abstraction. This array keeps a "distinguished index" whose corresponding entry is independent from the rest of the array. The `Distinguished_Index_Array` is useful to simplify the verification of concurrency properties of an implementation of the `Bounded_Queue` component described in previous work, or for an implementation of an array mapping algorithm.

**Listing 4.** Contract for `Distinguished_Index_Array`

```
Concept Distinguished_Index_Array_Template(type Entry);

  var Ids: powerset(Z);

  Type Family Distinguished_Index_Array is modeled by Cart_Prod
    Id: Z;
    Lower_Bound, Upper_Bound: Z;
    Contents: Z −> Entry;
    Distinguished_Index: Z;
```

```
  Domain: powerset(Z);
  Rest_In_Use: B;
end;
exemplar A;
  constraint A.Lower_Bound <= A.Upper_Bound and
    A.Id in Ids and A.Distinguished_Index in A.Domain;
  initialization ensures
    A.Lower_Bound = 0 and
    A.Upper_Bound = 0 and
    A.Id not in #Ids and
    A.Rest_In_Use = false and
    A.Domain = {A.Distinguished_Index};
end;

Operation Set_Bounds(evaluates LB, UB: Integer, updates A:
  Distinguished_Index_Array);
  requires LB <= UB and not A.Rest_In_Use;
  ensures A.Lower_Bound = LB and A.Upper_Bound = UB and
    not A.Rest_In_Use and
    A.Id not in #Ids and A.Id in Ids and A.Id /= #A.Id and
    A.Domain = {i: Z (A.Lower_Bound <= i and i < A.Upper_Bound)
   (i)};

Operation Swap_Distinguished_Entry(updates A:
  Distinguished_Index_Array, updates E: Entry);
  requires A.Lower_Bound <= A.Distinguished_Index and
    A.Distinguished_Index < A.Upper_Bound;
  ensures E = #A(A.Distinguished_Index) and A(A.
  Distinguished_Index) = #E and
    [[everything else about A stays the same]];

Operation Retrieve_Rest(updates A: Distinguished_Index_Array,
  replaces B: Distinguished_Index_Array);
  requires not A.Rest_In_Use;
  ensures A.Rest_In_Use and B.Domain = A.Domain \ {A.
  Distinguished_Index} and
    Eq_Except_On(A.Contents, B.Contents, A.Distinguished_Index)
  and
    B.Id = A.Id and B.Lower_Bound = A.Lower_Bound and B.
  Upper_Bound = A.Upper_Bound and
    not B.Rest_In_Use and
    [[everything else about A stays the same]];

Operation Replace_Rest(updates A: Distinguished_Index_Array,
  clears B: Distinguished_Index_Array);
```

```
    requires A.Rest_In_Use and B.Domain = A.Domain \ {A.
    Distinguished_Index};
    ensures not A.Rest_In_Use and
      Eq_Except_On(A.Contents, #B.Contents, A.Distinguished_Index)
     and
      [[everything else about A stays the same]];

  Operation Change_Distinguished_Index_To(updates A:
    Distinguished_Index_Array, evaluates i: Integer);
    requires not A.Rest_In_Use and not A.Distinguished_Entry_In_Use
    ;
    ensures A.Id = #A.Id and A.Contents = #A.Contents and
      A.Distinguished_Index = i and [[everything else about A
    stays the same]];

  Operation Lower_Bound(preserves A: Splittable_Array): Integer;
    ensures Lower_Bound = A.Lower_Bound;

  Operation Upper_Bound(preserves A: Splittable_Array): Integer;
    ensures Upper_Bound = A.Upper_Bound;

  Operation Is_In_Rest(preserves A: Distinguished_Index_Array,
    evaluates i: Integer): Boolean;
    ensures Is_In_Rest = (i in A.Domain and i /= A.
    Distinguished_Index);

  Operation Distinguished_Index(preserves A:
    Index_Partitionable_Array): Integer;
    ensures Distinguished_Index = A.Distinguished_Index;

  Operation Rest_Is_In_Use(preserves A: Distinguished_Index_Array):
     Boolean;
    ensures Rest_Is_In_Use = A.Rest_In_Use;

  Operation Ids_Match(preserves A1, A2: Distinguished_Index_Array)
    : Boolean;
    ensures Ids_Match = (A1.Id = A2.Id);

end Distinguished_Index_Array_Template;
```

**Listing 5.** An interference contract for `Distinguished_Index_Array_Template` which exposes the expected parallelism.

```
Interference Contract Distinguished_Index_Separate
  Detangles Distinguished_Index_Array_Template;
```

```
Partition for Distinguished_Index_Array is {Dist, Rest, Aux};

Operation Set_Bounds(evaluates LB, UB: Integer, updates A:
  Splittable_Array);
  affects A@Dist, A@Rest, A@Aux;

Operation Swap_Distinguished_Entry(updates A:
  Distinguished_Index_Array, replaces E: Entry);
  affects A@Dist;
  preserves A@Aux;

Operation Retrieve_Rest(updates A: Distinguished_Index_Array,
  replaces B: Distinguished_Index_Array);
  affects A@Rest;

Operation Replace_Rest(updates A: Distinguished_Index_Array,
  clears B: Distinguished_Index_Array);
  affects A@Rest;

Operation Change_Distinguished_Index_To(updates A:
  Distinguished_Index_Array, evaluates i: Integer);
  affects A@Aux;
  restructures A@Dist, A@Rest;

Operation Lower_Bound(preserves A: Splittable_Array): Integer;
  preserves A@Aux;

Operation Upper_Bound(preserves A: Splittable_Array): Integer;
  preserves A@Aux;

Operation Is_In_Rest(preserves A: Distinguished_Index_Array,
  evaluates i: Integer): Boolean;
  preserves A@Rest;

Operation Distinguished_Index(preserves A:
  Distinguished_Index_Array): Integer;
  preserves A@Aux;

Operation Rest_Is_In_Use(preserves A: Distinguished_Index_Array):
   Boolean;
  preserves A@Rest;

Operation Ids_Match(preserves A1, A2: Distinguished_Index_Array)
  : Boolean;
  preserves A1@Aux, A2@Aux;
```

```
end Distinguished_Index_Separate;
```

The **interference contract** for Distinguished_Index_Array is entirely straightforward. There are no conditional effects, so verifying non-interference in a **cobegin** statement is entirely syntactic. Moreover, the proofs discharged by using the concept in an implementation of another component are quite small.

This abstraction can be used to implement the Bounded_Queue concept described in previous work [1]. The interference contract Detanglement_No_3 is the most permissive of the interference contracts for Bounded_Queue_Template, and it permits concurrent execution of Enqueue with Dequeue, Dequeue_From_Long, or Swap_First_Entry when the queue is non-empty (and non-full). The queue might be used in a producer-consumer context when the queue is known never to be empty.

```
Interference Contract Detanglement_No_3
  Detangles BoundedQueue;

  Partition for Queue is (a, b);

  Operation EnQueue(updates Q: Queue, clears E: Entry);
    affects Q@b;
    affects E@*;
    when |Q| = 0:
      affects Q@a;

  Operation Dequeue(updates Q: Queue, replaces E: Entry);
    affects Q@a;
    affects E@*;
    restructures Q@a, Q@b;

  Operation Dequeue_From_Long(updates Q: Queue, replaces E: Entry)
    ;
    affects Q@a;
    affects E@*;
    restructures Q@a, Q@b;

  Operation Swap_First_Entry(updates Q: Queue, replaces E: Entry);
    affects Q@a;
    affects E@*;

  Operation Length(preserves Q: Queue): Integer;
    preserves Q@a, Q@b;

  Operation Rem_Capacity(preserves Q: Queue): Integer;
    preserves Q@a, Q@b;
```

```
end Detanglement_No_3;
```

```
Realization Distinguished_Index_Array_Realization
  Implements Bounded_Queue_Template;
  Respects Detanglement_No_3;
  Uses Distinguished_Index_Array_Template;

  function Fn_To_String_W_Wrapping(F: Z -> E, Start: Z, End: Z):
    string of E is
    if Start <= End then Iterated_Concatenation(i = Start..End, F
    (i))
    else Iterated_Concatenation(i = Start..(MAX_LENGTH + 1), F(i))
     o Iterated_Concatenation(i = 0..End, F(i));

  lemma |Fn_To_String_W_Wrapping(F, S, E)| = 0 iff S = E;

  Representation for Queue is
   (Contents: Distinguished_Index_Array,
    Post_Tail: Integer);
  Exemplar Q
    Convention
      0 <= Q.Post_Tail and Q.Post_Tail <= MAX_LENGTH and
      Q.Contents.Domain = {i: Z (0 <= i and i <= MAX_LENGTH) (i)
    };
    Correspondence function is
      Fn_To_String_W_Wrapping(Q.Contents.Contents, Q.Contents.
    Distinguished_Index, Q.Post_Tail);
    Interference Correspondence
      Q.Contents@Dist in Q@a,
      Q.Contents@Rest in Q@b,
      Q.Contents@Idx in Q@a,
      Q.Post_Tail in Q@b;
    Initialization
      Set_Bounds(Q.Contents, 0, MAX_LENGTH);
  end Queue;

  Operation Enqueue(updates Q: Queue, clears E: Entry);
    If (Is_In_Rest(Q, Q.Post_Tail))
      Var A: Distinguished_Index_Array;
      Retrieve_Rest(Q.Contents, A);
      Change_Distinguished_Index_To(Q.Post_Tail, A);
      Swap_Distinguished_Entry(A, E);
      Replace_Rest(Q.Contents, A);
    Else
```

```
      Swap_Distinguished_Entry(Q.Contents, E);
    end if;
    Q.Post_Tail := (Q.Post_Tail + 1) mod (MAX_LENGTH + 1);
    Clear(E);
  end Enqueue;

  Operation Dequeue(updates Q: Queue, replaces E: Entry);
    Swap_Distinguished_Entry(Q.Contents, E);
    Var Spec: Integer;
    Spec := Distinguished_Index(Q.Contents);
    Spec := (Spec + 1) mod (MAX_LENGTH + 1);
    Change_Distinguished_Index_To(Q.Contents, Spec);
  end Dequeue;

  Operation Dequeue_From_Long(updates Q: Queue, replaces E: Entry)
    ;
    Dequeue(Q, E);
  end Dequeue_From_Long;

  Operation Swap_First_Entry(updates Q: Queue, replaces E: Entry);
    Swap_Distinguished_Entry(Q.Contents, E);
  end Swap_First_Entry;

  Operation Length(preserves Q: Queue): Integer;
    Length := Q.Post_Tail − Distinguished_Index(Q.Contents);
    Length := Length mod (MAX_LENGTH + 1);
  end Length;

  Operation Rem_Capacity(preserves Q: Queue): Integer;
    Var Length: Integer;
    Length := Length(Q);
    Rem_Capacity := MAX_LENGTH − Length;
  end Rem_Capacity

end Distinguished_Index_Array_Realization;
```

To illustrate how to reason about an implementation in this system, we show reasoning tables for the `Enqueue` and `Dequeue` operations. These reasoning tables are abbreviated for space reasons. Excluded from the table are columns for reasoning only about *functional* correctness and detanglement facts and obligations about *oblivious*-mode partitions and those which are not mentioned in the interference contract of the operation.

First, a few notes about the tables. The *Detanglement Facts* in state 0 of both tables define the partition membership of `Q`, which is necessary for showing effects of statements on partitions of `Q`, given their effects on *fields* of `Q`. The *Path Condition* in the table for `Enqueue` comes directly from the postcondition of the

method in the if statement's condition. *Detanglement Obligations* are derived from the interference contract for the operation being implemented. In these cases, each line of the method has exactly the same detanglement obligations, but that is not the case in general. These obligations must be met by the interference contract of the following statement for the proof to proceed.

Table 1: Abbreviated Reasoning Table for the `Enqueue` operation for `Bounded_Queue` realized by `Special_Index_Array_Realization`.

| State | Code | Path Condition | Detanglement Facts | Detanglement Obligations |
|---|---|---|---|---|
| **0** | `Begin Enqueue(updates Q: Queue, clears E: Entry)` | | | |
| | | | $mem(Q@a) =$ $\{Q.Contents@Int,$ $Q.Contents@Idx\}$ $mem(Q@b) =$ $\{Q.Contents@Rest,$ $Q.Post\_Tail@*\}$ | $Q@b \trianglerighteq affects$ $E@* \trianglerighteq affects$ $\lvert Q\rvert = 0 \Rightarrow$ $Q@a \trianglerighteq affects$ |
| **1** | `If (Is_In_Rest(Q.Contents, Q.Post_Tail))` | | | |
| | | $Q.Post\_Tail_0 \in$ $Q.Contents.Domain_0 \wedge$ $Q.Post\_Tail_0 \neq$ $Q.Contents.Special\_Index_0$ | $Q.Contents@Rest =$ $preserves$ $Q.Post\_Tail =$ $preserves$ | $Q@b \trianglerighteq affects$ $E@* \trianglerighteq affects$ $\lvert Q\rvert = 0 \Rightarrow$ $Q@a \trianglerighteq affects$ |
| **2** | `Var A: Special_Index_Array;` `Retrieve_Rest(Q.Contents, A);` | | | |
| | | $Q.Post\_Tail_0 \in$ $Q.Contents.Domain_0 \wedge$ $Q.Post\_Tail_0 \neq$ $Q.Contents.Special\_Index_0$ | $Q.Contents@Rest =$ $affects$ | $Q@b \trianglerighteq affects$ $E@* \trianglerighteq affects$ $\lvert Q\rvert = 0 \Rightarrow$ $Q@a \trianglerighteq affects$ |
| **3** | `Change_Special_Index_To(Q.Post_Tail, A);` | | | |
| | | $Q.Post\_Tail_0 \in$ $Q.Contents.Domain_0 \wedge$ $Q.Post\_Tail_0 \neq$ $Q.Contents.Special\_Index_0$ | $Q.Post\_Tail@* =$ $preserves$ | $Q@b \trianglerighteq affects$ $E@* \trianglerighteq affects$ $\lvert Q\rvert = 0 \Rightarrow$ $Q@a \trianglerighteq affects$ |
| **4** | `Swap_Interesting_Entry(A, E);` | | | |
| | | $Q.Post\_Tail_0 \in$ $Q.Contents.Domain_0 \wedge$ $Q.Post\_Tail_0 \neq$ $Q.Contents.Special\_Index_0$ | | $Q@b \trianglerighteq affects$ $E@* \trianglerighteq affects$ $\lvert Q\rvert = 0 \Rightarrow$ |
| **5** | `Replace_Rest(Q.Contents, A);` | | | |
| | | $Q.Post\_Tail_0 \in$ $Q.Contents.Domain_0 \wedge$ $Q.Post\_Tail_0 \neq$ $Q.Contents.Special\_Index_0$ | $Q.Contents@Rest =$ $affects$ | $Q@b \trianglerighteq affects$ $E@* \trianglerighteq affects$ $\lvert Q\rvert = 0 \Rightarrow$ $Q@a \trianglerighteq affects$ |
| **6** | `Else` | | | |
| | | $Q.Post\_Tail_0 \notin$ $Q.Contents.Domain_0 \vee$ $Q.Post\_Tail_0 =$ $Q.Contents.Special\_Index_0$ | $Q.Contents@Rest =$ $preserves$ | $Q@b \trianglerighteq affects$ $E@* \trianglerighteq affects$ $\lvert Q\rvert = 0 \Rightarrow$ $Q@a \trianglerighteq affects$ |
| | `Swap_Interesting_Entry(A, E);` | | | |

Table 1: (continued)

| State | Code | Path Condition | Detanglement Facts | Detanglement Obligations |
|---|---|---|---|---|
| 7 | | $Q.Post\_Tail_0 \notin Q.Contents.Domain_0 \vee Q.Post\_Tail_0 = Q.Contents.Special\_Index_0$ | $Q.Contents@Int = affects$ | $Q@b \trianglerighteq affects$<br>$E@* \trianglerighteq affects$<br>$\|Q\| = 0 \Rightarrow Q@a \trianglerighteq affects$ |
| | `End If` | | | |
| 8 | | | | $Q@b \trianglerighteq affects$<br>$\|Q\| = 0 \Rightarrow Q@a \trianglerighteq affects$ |
| | `Q.Post_Tail := (Q.Post_Tail + 1) mod (MAX_LENGTH + 1);` | | | |
| 9 | | | $Q.Post\_Tail@* = affects$ | $Q@b \trianglerighteq affects$<br>$E@* \trianglerighteq affects$<br>$\|Q\| = 0 \Rightarrow Q@a \trianglerighteq affects$ |
| | `Clear(E);` | | | |
| 10 | | | $E@* = affects$ | |
| | `End Enqueue;` | | | |

Table 2: Abbreviated Reasoning Table for the `Dequeue` operation for `Bounded_Queue` realized by `Special_Index_Array_Realization`.

| State | Code | Path Condition | Detanglement Facts | Detanglement Obligations |
|---|---|---|---|---|
| | `Begin Dequeue(updates Q: Queue, replaces E: Entry)` | | | |
| **0** | | | $mem(Q@a) =$ $\{Q.Contents@Int,$ $Q.Contents@Idx\}$ $mem(Q@b) =$ $\{Q.Contents@Rest,$ $Q.Post\_Tail@*\}$ | |
| | `Swap_Interesting_Entry(Q.Contents, E);` | | | |
| **1** | | | $Q.Contents@Int =$ $affects$ $E@* = affects$ | $Q@a \trianglerighteq affects$ $E@* \trianglerighteq affects$ |
| | `Var Spec: Integer;` `Spec := Special_Index(Q.Contents);` | | | |
| **2** | | | $Q.Contents@Idx =$ $preserves$ | $Q@a \trianglerighteq affects$ $E@* \trianglerighteq affects$ |
| | `Spec := (Spec + 1) mod (MAX_LENGTH + 1);` | | | |
| **3** | | | | $Q@a \trianglerighteq affects$ $E@* \trianglerighteq affects$ |
| | `Change_Special_Index_To(Q.Contents, Spec);` | | | |
| **4** | | | $Q.Contents@Idx =$ $affects$ | $Q@a \trianglerighteq affects$ $E@* \trianglerighteq affects$ |
| | `End Dequeue;` | | | |

### 2.3 Index Partitionable Array

The third Array abstraction we present is a generalization of each of the two above abstractions. That is, each of the above concepts can be implemented on top of this more general concept.

**Listing 6.** Contract for `Index_Partitionable_Array`

```
Concept Index_Partitionable_Arary_Template(type Entry);

  var Ids: powerset(Z);
    initialization ensures Ids = empty_set;

  Type Family Index_Partitionable_Array is modeled by Cart_Prod
    Id: Z;
    Lower_Bound, Upper_Bound: Z;
    Contents: Z -> Entry;
    Black: powerset(Z);
    White: powerset(Z);
    Black_Part_In_Use: B;
    White_Part_In_Use: B;
  end;
    exemplar A;
      constraint A.Lower_Bound <= A.Upper_Bound and
        A.Id in Ids and
        A.Black intersect A.White = empty_set;
      initialization ensures
        A.Lower_Bound = 0 and A.Upper_Bound = 0 and
        not A.Black_Part_In_Use and
        not A.White_Part_In_Use and
        A.Id not in #Ids and
        A.White = {i: Z (A.Lower_Bound <= i and i < A.Upper_Bound
    ) (i)};
  end;

  Operation Set_Bounds(evaluates LB, UB: Integer; updates A:
    Index_Partitionable_Array);
    requires LB <= UB and not A.Black_Part_In_Use and
      not A.White_Part_In_Use;
    ensures A.Lower_Bound = LB and A.Upper_Bound = UB and
      not A.Black_Part_In_Use and not A.White_Part_In_Use and
      not A.Id in #Ids and not #A.Id in Ids and
      A.White union A.Black =
        {i: Z (A.Lower_Bound <= i and i < A.Upper_Bound) (i)};

  Operation Make_Entry_Black(evaluates p: Integer, updates A:
    Index_Partitionable_Array);
```

```
  requires not A.Black_Part_In_Use and not A.White_Part_In_Use
  and
    p in A.Black or p in A.White;
  ensures p in A.Black and not p in A.White and
    [[everything else about A stays the same]];

Operation Make_Entry_White(evaluates p: Integer, updates A:
  Index_Partitionable_Array);
  requires not A.Black_Part_In_Use and not A.White_Part_In_Use
  and
    p in A.Black union A.White;
  ensures p in A.White and not p in A.Black and
    [[everything else about A stays the same]];

Operation Swap_Entry_At(evaluates i: Integer, updates A:
  Index_Partitionable_Array, updates E: Entry);
  requires not A.Black_Part_In_Use and not A.White_Part_In_Use
  and
    A.Lower_Bound <= i and i < A.Upper_Bound and
    i in A.Black union A.White;
  ensures E = #A.Contents(i) and A.Contents(i) = #E and
    [[everything else about A stays the same]];

Operation Retrieve_Black_Part(updates A:
  Index_Partitionable_Array, replaces B:
  Index_Partitionable_Array);
  requires not A.Black_Part_In_Use;
  ensures A.Id = #A.Id and A.Black_Part_In_Use and B.Id = A.Id
  and
    B.Black = #A.Black and
    B.White = empty_set and
    B.Contents = #A.Contents and
    not B.Black_Part_In_Use and not B.White_Part_In_Use and
    [[everything else about A stays the same]];

Operation Replace_Black_Part(updates A: Index_Partitionable_Array
  , clears B: Index_Partitionable_Array);
  requires A.Black_Part_In_Use and
    not B.Black_Part_In_Use and not B.White_Part_In_Use and
    B.Id = A.Id and B.Black union B.White = A.Black;
  ensures A.Contents = #B.Contents and not A.Lower_Part_In_Use
  and
    [[everything else about A stays the same]];
```

```
Operation Retrieve_White_Part(updates A:
  Index_Partitionable_Array, replaces B:
  Index_Partitionable_Array);
  requires not A.White_Part_In_Use;
  ensures A.Id = #A.Id and A.White_Part_In_Use and B.Id = A.Id
  and
    B.White = #A.White and
    B.Black = empty_set and
    B.Contents = #A.Contents and
    not B.Black_Part_In_Use and not B.White_Part_In_Use and
    [[everything else about A stays the same]];

Operation Replace_White_Part(updates A: Index_Partitionable_Array
  , clears B: Index_Partitionable_Array);
  requires A.White_Part_In_Use and
    not B.Black_Part_In_Use and not B.White_Part_In_Use and
    B.Id = A.Id and B.Black union B.White = A.White;
  ensures A.Contents = #B.Contents and not A.Lower_Part_In_Use
  and
    [[everything else about A stays the same]];

Operation Lower_Bound(preserves A: Splittable_Array): Integer;
  ensures Lower_Bound = A.Lower_Bound;

Operation Upper_Bound(preserves A: Splittable_Array): Integer;
  ensures Upper_Bound = A.Upper_Bound;

Operation Black_Part_Is_In_Use(preserves A:
  Index_Partitionable_Array): Boolean;
  ensures Black_Part_Is_In_Use = A.Black_Part_In_Use;

Operation White_Part_Is_In_Use(preserves A:
  Index_Partitionable_Array): Boolean;
  ensures White_Part_In_Use = A.White_Part_In_Use;

Operation Is_White(evaluates i: Integer; preserves A:
  Index_Partitionable_Array): Boolean;
  ensures Is_White = (i in A.White);

Operation Is_Black(evaluates i: Integer; preserves A:
  Index_Partitionable_Array): Boolean;
  ensures Is_Black = (i in A.Black);

Operation Ids_Match(preserves A1, A2: Index_Partitionable_Array)
  : Boolean;
```

```
    ensures Ids_Match = (A1.Id = A2.Id);

end Index_Partitionable_Array_Template;
```

**Listing 7.** An interference contract for `Index_Partitionable_Array_Template` which exposes the expected parallelism.

```
Interference Contract Sep_Partitions
  Detangles Index_Partitionable_Array_Template;

  Partition for Index_Partitionable_Array is {Blk, Wht, Aux};

  Operation Set_Bounds(evaluates LB, UB: Integer; updates A:
    Splittable_Array);
    affects A@Blk, A@Wht, A@Aux;

  Operation Make_Entry_Black(evaluates p: Integer, updates A:
    Index_Partitionable_Array);
    affects A@Blk, A@Wht;

  Operation Make_Entry_White(evaluates p: Integer, updates A:
    Index_Partitionable_Array);
    affects A@Wht, A@Blk;

  Operation Swap_Entry_At(evaluates i: Integer, updates A:
    Index_Partitionable_Array, updates E: Entry);
    when i in A.Black
      affects A@Blk;
    when i in A.White
      affects A@Wht;

  Operation Retrieve_Black_Part(updates A:
    Index_Partitionable_Array, replaces B:
    Index_Partitionable_Array);
    affects A@Blk;

  Operation Replace_Black_Part(updates A: Index_Partitionable_Array
    , clears B: Index_Partitionable_Array);
    affects A@Blk;

  Operation Retrieve_White_Part(updates A:
    Index_Partitionable_Array, replaces B:
    Index_Partitionable_Array);
    affects A@Wht;
```

```
  Operation Replace_White_Part(updates A: Index_Partitionable_Array
    , clears B: Index_Partitionable_Array);
    affects A@Wht;

  Operation Lower_Bound(preserves A: Splittable_Array): Integer;
    preserves A@Aux;

  Operation Upper_Bound(preserves A: Splittable_Array): Integer;
    preserves A@Aux;

  Operation Black_Part_Is_In_Use(preserves A:
    Index_Partitionable_Array): Boolean;
    preserves A@Blk;

  Operation White_Part_Is_In_Use(preserves A:
    Index_Partitionable_Array): Boolean;
    preserves A@Wht;

  Operation Is_White(evaluates i: Integer; preserves A:
    Index_Partitionable_Array): Boolean;
    preserves A@Wht;

  Operation Is_Black(evaluates i: Integer; preserves A:
    Index_Partitionable_Array): Boolean;
    preserves A@Blk;

  Operation Ids_Match(preserves A1, A2: Index_Partitionable_Array)
    : Boolean;
    preserves A1@Aux, A2@Aux;

end Sep_Partitions;
```

Using Index−Partitionable_Array_Template, it is possible to realize both Splittable_Array_Template and Distinguished_Index_Array_Template, including respecting their permissive inteference contracts.

**Listing 8.** A realization of Splittable_Array_Template built using Index−Partitionable_Array_Template

```
Realization Index_Partitionable_Array_Realization
  Implements Splittable_Array_Template;
  Respects Upper_Lower_Separate;
  Uses Index_Partitionable_Array_Template;

  Representation for Splittable_Array is
   (Arr: Index_Partitionable_Array,
    Split_Point: Integer);
```

```
Exemplar A
  Convention
    A.Arr.Black = {i: Z (A.Arr.Lower_Bound <= i and i < A.
  Split_Point) (i)} and
    A.Arr.Black = {i: Z (A.Split_Point <= i and i < A.Arr.
  Upper_Bound) (i)};
  Correspondence
    conc.A.Id = A.Arr.Id and
    conc.A.Contents = A.Arr.Contents and
    conc.A.Lower_Bound = A.Arr.Lower_Bound and
    conc.A.Upper_Bound = A.Arr.Upper_Bound and
    conc.A.Split_Point = A.Split_Point and
    conc.A.Parts_In_Use =
      (A.Arr.Black_Part_In_Use and A.Arr.White_Part_In_Use);
  Interference Correspondence
    A.Arr@Blk in A@Low,
    A.Arr@Wht in A@Up,
    A.Arr@Idx in A@Aux,
    A.Split_Point in A@Aux;
end Splittable_Array;

Operation Set_Bounds(evaluates LB, UB: Integer; updates A:
  Splittable_Array);
  Set_Bounds(A.Arr);
end Set_Bounds;

Operation Set_Split_Point(evaluates i: Integer; updates A:
  Splittable_Array);
  if Is_Greater(A.Split_Point, i) then
    loop while (not Are_Equal(A.Split_Point, i))
      decreases
        A.Split_Point − i;
      maintains
        (A.Split_Point = A.Arr.Upper_Bound or (A.Split_Point +
  1) in A.Arr.White) and
        A.Arr.White \ #A.Arr.White = #A.Arr.Black \ A.Arr.Black
  and i = #i;
    do
      Decrement(A.Split_Point);
      Make_Entry_White(A.Split_Point, A);
    end loop;
  else
    loop while (not Are_Equal(A.Split_Point, i))
      decreases
        i − A.Split_Point;
```

```
      maintains
        (A.Split_Point = A.Arr.Lower_Bound or (A.Split_Point −
  1) in A.Arr.Black) and
        A.Arr.Black \ #A.Arr.Black = #A.Arr.White \ A.Arr.White
  and i = #i;
    do
      Make_Entry_Black(A.Split_Point, A);
      Increment(A.Split_Point);
    end loop;
  end if;
end Set_Split_Point;

Operation Swap_Entry_At(evaluates i: Integer, updates A:
  Splittable_Array, updates E: Entry);
  Swap_Entry_At(i, A.Arr, E);
end Swap_Entry_At;

Operation Split(updates A: Splittable_Array, replaces L, U:
  Splittable_Array);
  Retrieve_Black_Part(A.Arr, L);
  Retrieve_White_Part(A.Arr, U);
end Split;

Operation Combine(updates A: Splittable_Array, clears L, U:
  Splittable_Array);
  Replace_Black_Part(A.Arr, L);
  Replace_White_Part(A.Arr, U);
end Combine;

Operation Lower_Bound(preserves A: Splittable_Array): Integer;
  Lower_Bound := A.Lower_Bound;
end Lower_Bound;

Operation Upper_Bound(preserves A: Splittable_Array): Integer;
  Upper_Bound := A.Upper_Bound;
end Upper_Bound;

Operation Split_Point(preserves A: Splittable_Array): Integer;
  Split_Point := A.Split_Point;
end Split_Point;

Operation Parts_Are_In_Use(preserves A: Splittable_Array):
  Boolean;
  Parts_Are_Is_In_Use := Black_Part_Is_In_Use(A.Arr) and
  White_Part_Is_In_Use(A.Arr);
```

```
  end Lower_Part_Is_In_Use;

  Operation Ids_Match(preserves A1, A2: Splittable_Array): Boolean
    ;
    Ids_Match := Ids_Match(A1, A2);
  end Ids_Match;

end Index_Partitionable_Array_Realization;
```

**Listing 9.** A realization of `Distinguished_Index_Array` built using `Index_Partitionable_Array_Template`

```
Realization Index_Partitionable_Array_Realization
  Implements Distinguished_Index_Array;
  Respects Distinguished_Index_Separate;
  Uses Index_Partitionable_Array_Template;

  Representation for Distinguished_Index_Array is
   (Arr: Index_Partitionable_Array,
    Int: Index_Partitionable_Array,
    Distinguished_Index: Integer);
  Exemplar A
    Convention
      A.Arr.Black = {A.Distinguished_Index};
    Correspondence
      conc.A.Id = A.Arr.Id and
      conc.A.Lower_Bound = A.Arr.Lower_Bound and
      conc.A.Upper_Bound = A.Arr.Upper_Bound and
      conc.A.Contents = A.Arr.Contents and
      conc.A.Domain = A.Arr.Black union A.Arr.White and
      conc.A.Distinguished_Index = A.Distinguished_Index and
      conc.A.Distinguished_Entry_In_Use = A.Arr.Black_Part_In_Use
    and
      conc.A.Rest_In_Use = A.Arr.White_Part_In_Use;
    Interference Correspondence
      A.Arr@Blk in A@Dist,
      A.Arr@Wht in A@Rest,
      A.Arr@Idx in A@Idx,
      A.Distinguished_Index in A@Idx;
      A.Int in A@Int;
  end Distinguished_Index_Array;

  Operation Set_Bounds(evaluates LB, UB: Integer; updates A:
    Splittable_Array);
    Set_Bounds(A.Arr);
  end Set_Bounds;
```

```
Operation Swap_Distinguished_Entry(updates A:
  Distinguished_Index_Array; updates E: Entry);
  Retrieve_Black_Part(A.Arr, A.Int);
  Swap_Entry_At(A.Distinguished_Index, A.Int, E);
  Replace_Black_Part(A.Arr, A.Int);
end Retrieve_Distinguished_Entry;

Operation Retrieve_Rest(updates A: Distinguished_Index_Array;
  replaces B: Distinguished_Index_Array);
  Retrieve_White_Part(A.Arr, B);
end Retrieve_Rest;

Operation Replace_Rest(updates A: Distinguished_Index_Array;
  clears B: Distinguished_Index_Array);
  Replace_White_Part(A.Arr, B);
end Replace_Rest;

Operation Change_Distinguished_Index_To(evaluates i: Integer;
  updates A: Distinguished_Index_Array);
  Make_Entry_White(A.Distinguished_Index, A.Arr);
  A.Distinguished_Index := i;
  Make_Entry_Black(A.Distinguished_Index, A.Arr);
end Change_Distinguished_Index_To;

Operation Lower_Bound(preserves A: Splittable_Array): Integer;
  Lower_Bound := A.Lower_Bound;
end Lower_Bound;

Operation Upper_Bound(preserves A: Splittable_Array): Integer;
  Upper_Bound := A.Upper_Bound;
end Upper_Bound;

Operation Is_In_Rest(preserves A: Distinguished_Index_Array,
  evaluates i: Integer): Boolean;
  Is_In_Rest := Is_White(A.Arr, i);
end Is_In_Rest;

Operation Distinguished_Index(preserves A:
  Distinguished_Index_Array): Integer;
  Distinguished_Index := A.Distinguished_Index;
end Distinguished_Index;

Operation Rest_Is_In_Use(preserves A: Distinguished_Index_Array):
   Boolean;
```

```
    Rest_Is_In_Use := White_Part_Is_In_Use(A.Arr);
  end White_Part_Is_In_Use;

  Operation Ids_Match(preserves A1: Distinguished_Index_Array, A2:
     Distinguished_Index_Array): Boolean;
    Ids_Match := (A1.Id = A2.Id);

end Index_Partitionable_Array_Realization;
```

**Realization of `Index_Partitionable_Array`** Although the two sub-arrays into which an `Index_Partitionable_Array` is split are reasoned about as if they were totally separate arrays by virtue of clean semantics, a reasonable implementation of this concept would *not* make any copies of the array. The interface for this component was designed with a shared implementation in mind so that a realization could employ an underlying (traditional) array that is shared among all `Index_Partitionable_Array` instances with the same `Id`. This careful design manifests itself in the use of the `Retrieve` and `Replace` operations as pseudo-synchronization points by flipping a boolean and preventing access to the array while either the black or white parts are in use. Doing so ensures that at any time, there is only one array with each `Id` that can access any given index. In this way, a realization can share an underlying array among instances with the same `Id` without introducing any interference. Enabling such a shared implementation is important for preserving performance benefits that programmers expect from parallel software.

**Acknowledgments**

We thank several members of our research groups at Clemson and Ohio State who have contributed to the discussions on topics contained in this paper. This research is funded in part by US NSF grants CCF-1161916 and DUE-1022941. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the NSF.

## References

1. A. Weide, P. A. G. Sivilotti, and M. Sitaraman. Enabling modular verication of concurrent programs with abstract interference contracts. Technical Report RSRG-16-05, Clemson University - School of Computing, December 2016.