Enabling Modular Verification of Concurrent Programs with Abstract Interference Contracts

AlanWeide, Paolo A. G. Sivilotti, and Murali Sitaraman

Technical Report RSRG-16-05
School of Computing
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

December 2016

Enabling Modular Verification of Concurrent Programs with Abstract Interference Contracts

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. When concurrent threads of execution do not modify shared data, their parallel execution is trivially equivalent to their sequential execution. For many imperative programming languages, however, the modular verification of this independence is often frustrated by (i) the possibility of aliasing between variables mentioned in different threads, and (ii) the lack of abstraction in the description of read/write effects of operations on shared data structures.
We describe a specification and verification framework in which abstract specifications of functional behavior are augmented with abstract interference effects that permit verification of client code with concurrent calls to operations of a data abstraction. To illustrate the approach, we present a classic concurrent data abstraction: the bounded queue. Three different implementations are described, each with different degrees of entanglement and hence different degrees of possible synchronization-free concurrency.

1 Introduction

Parallel programming is important for both large-scale high-performance systems and, increasingly, small-scale multi-core commodity software. Programming with multiple threads, however, is error-prone. Furthermore, when errors are made, they can be difficult to debug and correct because parallel programs are often nondeterministic. Non-trivial parallel programs designed with software engineering consideration will be invariably composed from reusable components, often ones that encapsulate data abstractions.

Given this context, the specific objective of this paper is to propose a specification and verification framework to guarantee entanglement- and lock-free execution of concurrent code that invokes operations on data abstractions. Guaranteeing, simultaneously, modularity of the verification process and the independence of concurrent threads is complicated by two key problems. The first of these is the possibility of aliasing between objects involved in different threads. The second problem concerns guaranteeing safe parallel execution of data abstraction operations on an object without violating abstraction.

At the core of a solution to the aliasing problem is a notion of *clean* operation calls whereby effects of calls are restricted to objects that are explicit parameters or to global objects that are explicitly specified as affected. Under this notion, regardless of the level of granularity, syntactically independent operation calls are always safe to parallelize. Clean semantics are at the core of the RESOLVE language and we take advantage of the benefits of such a notion in our specification and verification framework.

To illustrate our proposed solution to the second problem, we present a bounded queue data abstraction and describe in detail three different implementations (and briefly mention a fourth) that vary in their potential for parallelism among different queue operations. The data abstraction specification is typical, except that it is designed to avoid unintended aliasing. To capture the parallel potential in a class of implementations we augment the data abstraction specification with an interference specification which contains a description of an object's state (which might be substantially different from the description provided by the data abstraction specification), called an "interference contract", designed to facilitate guarantees of safe execution of concurrent client code. The second-level specification is typically still quite abstract and is devoid of concrete implementation details.

A client of such a parallelized component identifies which interference specification to use. By doing so, the client is able to determine which operations it can execute in parallel from its code while remaining oblivious to implementation details. Furthermore, it can be verified that this code is safe to parallelize given the interference specification used by the client.

Inside of a realization, an implementer must define a mapping from the concrete model to the interference model defined in the interference contract. This mapping is entirely analogous to the correspondence that defines a mapping from the concrete model to the abstract specification's model.

The novelty of the proposed solution is that it modularizes the verification problem along abstraction boundaries. Specifically, verification of implementation code with respect to both its data abstraction and interference specification is done once in the lifetime of the implementation. Verification of client code relies strictly on the specifications.

2 Related Work

Classical solutions to the interference problem (e.g., [4]) would involve defining and using locks, but neither the solutions nor the proofs of absence of interference here involve abstraction or specification. Lock-free solutions built using atomic read-write-modify primitives (e.g., compare-and-swap) allow finer granularity of parallelism, but the proofs of serializability in that context are often not modular and do not involve complex properties.

The objective of modular verification is widely shared. The work in [1], for example, involves specifying interference points. For data abstractions, the interference points would be set at the operation level, meaning two operations

may not execute concurrently on an object, even if they are disentangled at a "fine-grain" level. The work by Rodriguez, et al [9] to extend JML for concurrent code makes it possible to specify methods to be atomic through locking and other properties. Using JML* and a notion of dynamic frames, the work in [8] address safe concurrent execution in the context of more general solutions to address aliasing and sharing for automated verification. The work in [11] makes it possible to specify memory locations that fall within the realm of an object's lock. Chalice allows specification of various types of permissions and includes a notion of permission transfer [7]. Using them, it is possible to estimate an upper bound on the location sets that may be affected by a thread in Chalice.

Perhaps the most relevant work to this text is Deterministic Parallel Java (DPJ) [2], an extension of the Java programming language with specific constructs to facilitate the development and static analysis of concurrent programs. DPJ relies on the idea of *regions*, which are distinct parts of the memory heap. Each class-level variable may be placed into a region (if no region is specified, the variable is in a special region called "root"), and each class method specifies which regions are written to or read from by its execution. These "method effect summaries" allow the DPJ compiler to prove, statically, that every valid DPJ program exhibits deterministic input-output, regardless of any concurrency introduced throughout the program[3]. It's important to distinguish DPJ's "verification" from the verification of functional correctness that is performed by other work, such as RESOLVE. The DPJ compiler guarantees only that for any given input to a valid program (absent shared state), the output will be the same for every execution of that program.

Further related work by the ParaSail team in [12] combines the ideas of clean semantics and deterministic parallelism. The ParaSail team has modified the Ada programming language so that the compiler can verify, like DPJ, that a program exhibits deterministic input-output behavior. Again, there is no notion of proving functional correctness, but ParaSail–by virtue of leveraging clean semantics–can forego any additional programming constructs such as regions. ParaSail does not have pointers, and hence no aliases exist. Therefore, a conflict exists only if two method calls reference the same variable and this variable is written by either method. As with DPJ, the language offers guarantees of determinism, rather than a mechanism for functional verification.

3 The Clean Premise

At the core of the RESOLVE project is the notion of clean semantics [6]. In the RESOLVE programming language, there are no aliases. If two variables have dif-

[3] There are some limitations to DPJ's ability to guarantee determinisim. For example, it is possible to write a compilable DPJ program which uses a concurrent algorithm to construct a tree in which each node gets the value of some static variable (of the Node class) which is incremented with every call to the Node's constructor. This tree will not necessarily have the same values at the same nodes after each execution of this program.

ferent names, the programmer and verification engine can be certain that they refer to different objects. The interference specifications presented here rely heavily on this assumption in several key ways. The first is on the implementation side. By assuming clean semantics, the verification that an implementation satisfies an interference specification is reduced to a mostly syntactic problem. That is, a method does not read or write a variable whose name does not appear in the method body. For interference specifications that include conditional effects, this check is not purely syntactic. In these cases, the verification that a method respects its interference specification entails proving various verification conditions, similar to what is done for functional verification. More details about the verification of implementations are presented in Section 6.3. A second, related, role of clean semantics is on the client side. Concurrent method calls that name different arguments are independent. This check is purely syntactic. In cases where the concurrent calls name the same argument–for example, operations on a shared data structure–the check is not purely syntactic. Again, the verification of independence in this situation leverages the machinery of functional verification. The verification methodology is elaborated in Section 6.2.

4 The Idea: Interference Specifications

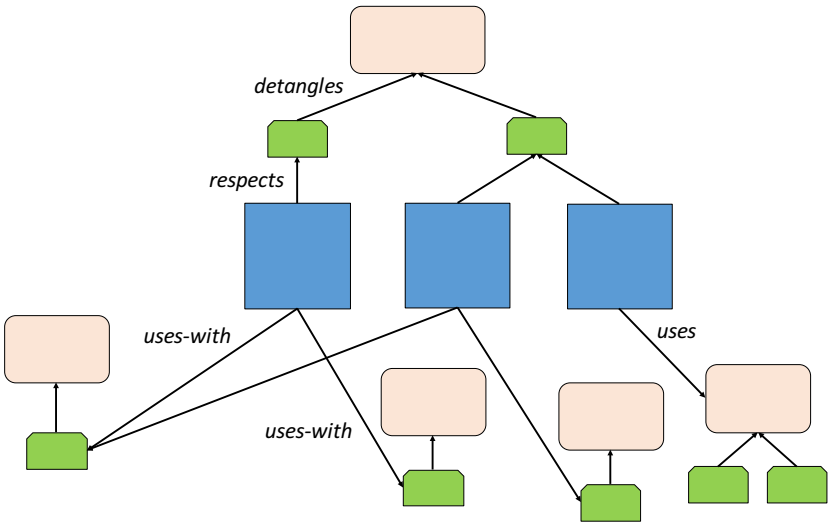

Fig. 1. A generic class diagram illustrating where interference specifications fit in the hierarchy of programming constructs in the RESOLVE programming language. The orange rounded rectangles are abstract specifications, the blue rectangles are realizations and the green truncated rectangles are interference contracts.

An *interference specification* describes how methods interact with one another and, therefore, under what conditions they may be safely parallelized. The key idea of interference specifications is that of *partitions*. Partitions are typeless entities which correspond to some part(s) of the concrete representation of an object. This mapping from concrete representation to partitions is defined as part of the implementation.

In addition to partitions, an interference specification provides *method effect summaries* which define the permissions a method has on each partition of its parameters: one of oblivious to, preserves, or affects. A method which is oblivious to a partition may neither read from nor write to any parts of the concrete state within that partition. A method that preserves a partition may read the values of the concrete fields in that partition, but not write to them. Finally, a method that affects a partition may read and write to any or all of the concrete fields within that partition. If a partition is not mentioned in an effect summary, that partition is **oblivious to** mode. Verifying that a method's implementation satisfies its effect summary and that client code using the method in a cobegin statement is doing so in a safe way is discussed in detail later. However, it should be noted that proving effect summaries on both the client and implementer sides is a purely syntactic problem when effect summaries are unconditional.

To increase the expressiveness of our framework, method effect summaries may be conditional on the *abstract state* of the object. Conditional effects allow a range of behaviors that are not possible with only unconditional effect summaries. They are denoted by when clauses in the effects and provide the implementer with some flexibility regarding the conditions under which the method may change the values of some fields in the concrete state. Conditional effects are novel because they provide a mechanism whereby arbitrarily complicated effect summaries may be constructed and be used in the verification that the realization code is correct.

5 Bounded Queue Data Abstraction Specification and Alternative Implementations

As an illustration of our approach to modular verification of concurrency, this section presents a classic concurrent data structure: the bounded queue. The first subsection presents the client view of the queue, with an abstract specification of the operations permitted on this data structure. The next three subsections each present a different implementation of this same specification. Each implementation permits different degrees of safe concurrency.

5.1 Abstract Specification

Here we present a RESOLVE-style specification for a bounded queue [10].

Abstractly, a queue is modeled as a mathematical string of items. The BoundedQueueTemplate concept defines operations, such as Enqueue, Dequeue, SwapFirstEntry, Length, and RemCapacity. These operations have been designed and

specified to avoid aliasing that arises when queues contain non-trivial objects and to facilitate clean semantics [5]. In addition to these operations, a swap operator is defined on all types to facilitate data exchange without deep or shallow copying [3]. Since this paper is concerned mainly with the challenge of concurrent execution of operations, we omit further discussion of this idea.

operation Enqueue (**alters** e: Item, **updates** q: Queue)
requires |q| < MAX_LENGTH
ensures q = #q * <#e>

The contract for Enqueue says several things. The requires clause says that in order to be called, there must be space left in the queue to put the new element (|q| < MAX_LENGTH). The ensures clause says that the outgoing value of q is the concatenation of the incoming ("old") value of q with the string containing the old value of e. Less formally, Enqueue puts e at the end of the queue.

operation Dequeue (**replaces** e: Item, **updates** q: Queue)
requires q /= empty_string
ensures #q = <e> * q

The requires clause says that in order to be called, q must not be empty. The ensures clause effectively says that the resulting element e and outgoing value of q, when concatenated, are the same as the original value of q.

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

The SwapFirstEntry operation makes it possible to retrieve or update the first entry, without introducing aliasing. The substring operation is denoted by the asymmetric [ ) parentheses. For a string s, the substring s[i, j) is inclusive on the left and exclusive on the right. For example, the string #q[0,1) is the string consisting of a single element, the first element in the string #q.

operation Length (**restores** q: Queue) : Integer
ensures Length = |q|

operation RemCapacity (**restores** q: Queue) : Integer
ensures RemCapacity = MAX_LENGTH − |q|

Length returns an integer equal to the number of elements in the queue. RemCapacity returns an integer equal to the number of free slots left in the queue before it becomes full.

5.2 Bounded Queue with Concurrent Enqueue/SwapFirstEntry (#1)

Implementation. A bounded queue can be implemented using an array and a pointer to the location of the head of the queue. The length of the queue is

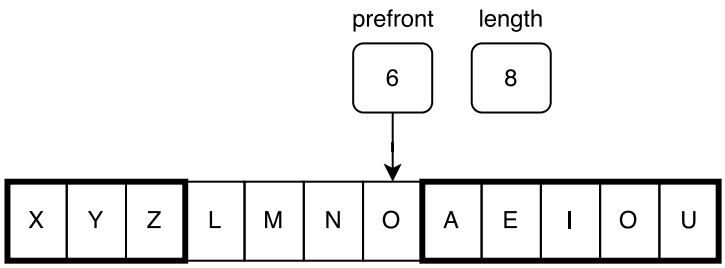

Fig. 2. An implementation of a bounded queue using an Array and two Integer fields, prefront and length.

also kept. Such an implementation is represented in Figure 2. The code listing is given in Listing 1.1.

```
realization ArrayWithLength
  implements BoundedQueue
  respects detanglement #1

  uses ArrayAsStringTemplate with IndependentSlots
  uses Mod for UnboundedIntegerFacility
  uses Increment for UnboundedIntegerFacility
  uses Add for UnboundedIntegerFacility

  facility ArrayFacility is ArrayAsStringTemplate(Item)

  type representation for Queue is (contents: Array,
                                    preFront: Integer,
                                    length: Integer)
    exemplar q
    convention q.contents.ub = MAX_LENGTH − 1 and
               q.contents.lb = 0 and
               0 <= q.preFront and
               q.preFront < MAX_LENGTH and
               0 <= q.length and
               q.length <= MAX_LENGTH
    correspondence function CIRCULAR_STRING_VALUE(contents.s,
      preFront, length)
    initialization
      ...
    interference correspondence
      q.preFront in q@i
      q.length in q@b
      q.contents@c[q.preFront + 1 mod MAX_LENGTH, q.preFront + 2 mod
      MAX_LENGTH) in q@a
```

```
      q.contents@c except on {q.preFront + 1 mod MAX_LENGTH} in q@b
30
  -- implicitly everything not part of interference correspondence must be
     preserved
  -- so:  q.contents@b is preserved in every method

  end Queue
35
  procedure Enqueue (clears e: Item, updates q: Queue)
    variable temp: Integer
    Increment(q.length)
    temp := Replica(q.preFront)
40  Add(temp, q.length)
    Mod(temp, maxLength)
    SwapItem(q.contents, temp, e)
    Clear(e)
  end Enqueue
45
  procedure Dequeue (replaces r: Item, updates q: Queue)
    Increment(q.preFront)
    Mod(q.preFront, maxLength)
    SwapItem(q.contents, q.preFront, r)
50  Decrement(q.length)
  end Dequeue

  procedure SwapFirstEntry (updates e: Item, updates q: Queue)
    variable temp: Integer
55  temp := Replica(q.preFront)
    Increment(temp)
    Mod(temp, maxLength)
    SwapItem(q.contents, temp, e)
  end SwapFirstEntry
60
end ArrayWithLength
```

Listing 1.1. The code for the implementation in Figure 2

Most of this is standard RESOLVE code. That is, it contains a convention, correspondence, initialization, and the code to implement the various operations that might be performed on an object of type Queue. The correspondence function, STRING_VALUE_SENTINEL, is defined in the abstract specification. The novelty lies in the **interference correspondence**, which as discussed above is a description of which parts of the concrete state map to which partitions of the interference contract being satisfied by this realization. In this case, q's *preFront* field resides in the *a* partition of q, and q's *postTail* field resides in the *b* partition.

The other two lines in the interference correspondence expose how layering components (see section 6.3) works in our framework. This realization uses ArrayAsStringTemplate, and in particular the IndependentSlots interference contract of ArrayAsStringTemplate (this interference contract can be found in the appendix). The *c* partition of the q.contents Array is a segmented partition, meaning that it is modeled as a *string of partition* and can be accessed just like any other string in RESOLVE. Thus, q.contents@c[head, head+1) is the substring of q.contents@c starting at index *head* and ending at the item before index *head+1* (that is, the string of length one containing the partition at index head of q.contents@c. The *except on* function returns the set of items in the string except for those at the indices which are elements of the second operator. So, from the interference correspondence most of q.contents@c is in partition q@a while the partition at index head in q.contents@c is in partition q@b. Upon examining the interference contract IndependentSlots for ArrayAsStringTemplate, it is apparent that this partitioning affords us the luxury of safely accessing different indices of the array in parallel.

Interference Specification. The interference specification which is satisfied by ArrayWithLength follows.

```
interference contract #1 detangles BoundedQueue

  partition for Queue is (a, b, i)

  procedure Enqueue (clears e: Item, updates q: Queue)
    affects q@b
    preserves q@i
    when |q| = 0 affects q@a

  procedure Dequeue (replaces r: Item, updates q: Queue)
    affects q@a, q@b, q@i

  procedure SwapFirstEntry (updates e: Item, updates q: Queue)
    affects q@a
    preserves q@i

end interference contract #1
```

Listing 1.2. The interference contract satisfied by the implementation in Listing 1.1

This interference contract exposes enough independence that a client may concurrently execute Enqueue and SwapFirstEntry, but not Enqueue and Dequeue. It divides the representation into 3 fields (called "q@a", "q@b", and "q@i"). Field q@a is *affected* by Dequeue and SwapFirstEntry all the time, and it is affected by Enqueue only when $|q| = 0$. Field q@b is affected by Enqueue and Dequeue in all possible cases, so Enqueue and Dequeue may not be called concurrently. Because Enqueue and SwapFirstEntry both *preserve* q@i, that effect

will not preclude the two operations from being executed safely in parallel. Furthermore, because the precondition for SwapFirstEntry (found in the abstract specification) requires that $|q| > 0$, if Enqueue and SwapFirstEntry are called in parallel, it cannot be the case that $|q| = 0$, and so the Enqueue method will never affect q@a, so Enqueue and SwapFirstEntry may always safely be executed in parallel. Recall that the default mode for an interference field is *oblivious to*, so SwapFirstEntry is oblivious to q@b, which Enqueue affects, and Enqueue is oblivious to q@a when $|q| > 0$, which SwapFirstEntry affects (and SwapFirstEntry may only be called when $|q| > 0$).

5.3 Bounded Queue with Concurrent Enqueue/Dequeue (#2)

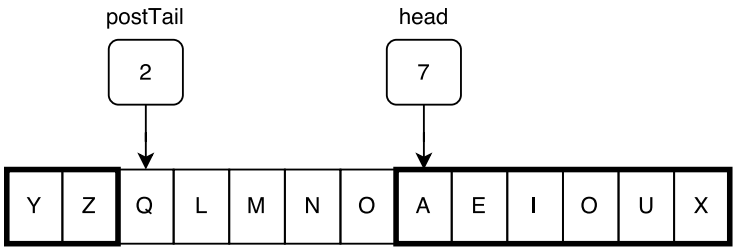

Fig. 3. An implementation of a bounded queue using an Array and two Integer fields, head and postTail, and a sentinel node to distinguish between a full queue and an empty one

Implementation. The code for the implementation in Figure 3 is in Listing 1.3.

```
realization ArrayWithSentinel
  implements BoundedQueue
  respects dentanglement #2

  uses ArrayAsStringTemplate with IndependentSlots
  uses Mod for UnboundedIntegerFacility
  uses Increment for UnboundedIntegerFacility

  facility ArrayFacility is ArrayAsStringTemplate(Item)

  type representation for Queue is (contents: Array,
                                    head: Integer,
                                    postTail: Integer)
    exemplar q
```

```
    convention
      0 <= q.head and q.head <= MAX_LENGTH and
      0 <= q.postTail and
      q.postTail <= MAX_LENGTH and
      q.contents.lb = 0 and
      q.contents.ub = MAX_LENGTH
    correspondence function STRING_VALUE_SENTINEL(q.contents, q.
      head, q.postTail)
    initialization
      variable zero: Integer
      variable maxLength: Integer
      maxLength := MaxLength()
      SetBounds(q.contents, zero, maxLength)

    interference correspondence
      q.head in q@a
      q.postTail in q@b
      q.contents@c[q.head, q.head+1) in q@a
      q.contents@c except on {q.head} in q@b

  end Queue

  procedure Enqueue (clears e: Item, updates q: Queue)
    SwapItem(q.contents, q.postTail, e)
    Clear(e)
    Increment(q.postTail)
    Mod(q.postTail, maxLength)
  end Enqueue

  procedure Dequeue (replaces r: Item, updates q: Queue)
    SwapItem(q.contents, q.head, r)
    Increment(q.head)
    Mod(q.head, maxLength)
  end Dequeue

  procedure SwapFirstEntry (updates e: Item, updates q: Queue)
    SwapItem(q.contents, q.head, e)
  end SwapFirstEntry

end ArrayWithSentinel
```

Listing 1.3. The code for the implementation in Figure 3

This realization uses an Array of length MAX_LENGTH + 1, and two Integer fields, head and postTail. The Array has a "sentinel" node so that an empty queue looks different than a full queue (otherwise both would satisfy the property

head = postTail). The head field is in interference field q@a, postTail is in q@b, and, like in the first implementation, q.contents@c is split between q@a and q@b.

Interference Specification. The interference contract which is satisfied by the ArrayWithSentinel implementation is below.

```
interference contract #2 detangles BoundedQueue

  partition for Queue is (a, b)

  procedure Enqueue (clears e: Item, updates q: Queue)
    affects q@b
    when |q| = 0
      affects q@a
    --when |q| > 0
      --oblivious to q@b

  procedure Dequeue (replaces r: Item, updates q: Queue)
    affects q@a

  procedure SwapFirstEntry (updates e: Item, updates q: Queue)
    affects q@a
    --oblivious to q@b

end interference contract #2
```

Listing 1.4. The inteference contract satisfied by the implementation in Listing 1.1

The second interference contract for the Bounded Queue specified above expresses the most available parallelism: given a non-empty queue, a client can call Enqueue and Dequeue in parallel, or Enqueue and SwapFirstEntry, but not Dequeue and SwapFirstEntry (the reason for this should be obvious). This interference specification divides the representation of the queue into two partitions, in this case called a and b. The effects summary for Enqueue states that it will always affect (that is, write to) the part of the representation that lies in partition b, and it will affect the part of the representation that lies in partition b only when the *abstract state* of the queue is such that there is at least one item in the queue. It's important to note that the conditional effects clauses may only depend on the abstract state of the object and not the concrete state because it may be the case that many realizations all satisfy the same interference contract. Recall that the default mode of a partition in an effect summary is *oblivious to*.

5.4 Bounded Queue with Extra Method to Provide Concurrency (#3)

Implementation. A discussion of the implementation described by Figure 4 is below.

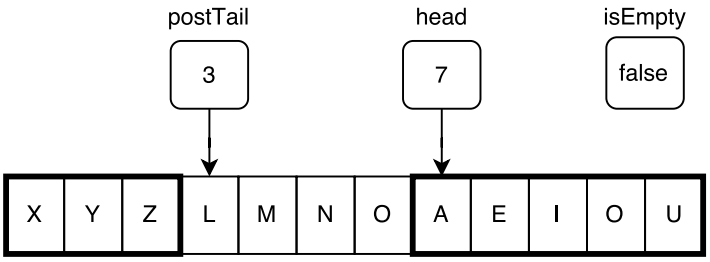

Fig. 4. An implementation of a bounded queue using an Array and two Integer fields, head and postTail, and a boolean flag to distinguish between a full queue and an empty one

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

Listing 1.5. An extra method to provide concurrency for the implementation described by Firgure 4

We add an extra method, DequeueFromLong, to provide safe concurrency between Dequeue and Enqueue. As seen in the implementation code below, the Dequeue operation requires a check on q.postTail so that it may set q.isEmpty as appropriate. Unfortunately for concurrency, the Enqueue method must also set that value. The precondition for DequeueFromLong (**requires** |q| >= 2) guarantees that the queue will never be empty after a call to this method, and thus does not need to read the value of q.postTail.

```
realization ArrayWithFlag
  implements BoundedQueue
  respects dentanglement #3

  uses ArrayAsStringTemplate with IndependentSlots
  uses Mod for UnboundedIntegerFacility
  uses Increment for UnboundedIntegerFacility

  facility ArrayFacility is ArrayAsStringTemplate(Item)

  type representation for Queue is (contents: Array,
                                    head: Integer,
                                    postTail: Integer,
                                    isEmpty: Boolean)
    exemplar q
    convention 0 <= q.head and q.head <= MAX_LENGTH and
               0 <= q.postTail and
               q.postTail < MAX_LENGTH and
               q.contents.lb = 0 and
```

```
20            q.contents.ub = MAX_LENGTH − 1
  correspondence function STRING_VALUE(q.contents, q.head, q.
    postTail)
  initialization
    variable zero: Integer
    variable maxLength: Integer
25  maxLength := MaxLength()
    SetBounds(q.contents, zero, maxLength)

  interference correspondence
    q.head in q@a
30  q.postTail in q@b
    q.contents@c[q.head, q.head+1) in q@a
    q.contents@c except on {q.head} in q@b
    q.isEmpty in q@m

35 end Queue

  procedure Enqueue (clears e: Item, updates q: Queue)
    SwapItem(q.contents, q.postTail, e)
    Clear(e)
40  Increment(q.postTail)
    Mod(q.postTail, maxLength)
    q.isEmpty := false
  end Enqueue

45 procedure Dequeue (replaces r: Item, updates q: Queue)
    SwapItem(q.contents, q.head, r)
    Increment(q.head)
    Mod(q.head, maxLength)
    if (q.head = q.postTail)
50    q.isEmpty := true;
  end Dequeue

  procedure DequeueFromLong (replaces r: Item, updates q: Queue)
    SwapItem(q.contents, q.head, r)
55  Increment(q.head)
    Mod(q.head, maxLength)
  end DequeueFromLong

  procedure SwapFirstEntry (updates e: Item, updates q: Queue)
60  SwapItem(q.contents, q.head, e)
  end SwapFirstEntry
```

end realization ArrayWithFlag

Listing 1.6. The implementation described by Firgure 4

This realization is identical in most respects to #2, but instead of a sentinel node in the array to distinguish between an empty and full array, the array is exactly MAX_LENGTH items long and there is a boolean flag, q.isEmpty, which is true exactly when |q| = 0. This boolean flag is in a field on its own, q@m.

Interference Specification The interference contract satisfied by ArrayWithFlag is below.

```
interference contract #2 detangles BoundedQueue

  partition for Queue is (a, b, m)

  procedure Enqueue (clears e: Item, updates q: Queue)
    affects q@b
    when |q| = 0
      affects q@a, q@m
    otherwise
      preserves q@m

  procedure Dequeue (replaces r: Item, updates q: Queue)
    affects q@a
    when |q| = 1
      affects q@b, q@m
    otherwise
      preserves q@b

  procedure DequeueFromLong (replaces r: Item, updates q: Queue)
    affects q@a
    -- oblivious to q@b, q@m

  procedure SwapFirstEntry (updates e: Item, updates q: Queue)
    affects q@a

end interference contract #2
```

Listing 1.7. The interference contract satisfied by the implementation in Listing 1.6

This interference contract expresses some parallelism (i.e., between Enqueue and SwapFirstEntry), but requires the addition of an extra method, DequeueFromLong, to fully exploit its potential. Because Dequeue *preserves* q@b in cases where |q| > 1 (and affects it otherwise), Enqueue cannot safely execute in parallel with it because Enqueue *affects* q@b in all cases. The DequeueFromLong operation, which requires |q| > 1 allows an implementation to forego a check of the concrete fields that are part of q@b and thus be *oblivious to* q@b. Therefore,

in any state where Enqueue and DequeueFromLong can be called in parallel, that concurrency is safe.

6 A High-Level, Conceptual View of Modular Verification Using Interference Specifications

6.1 Modular Verification in General

There are some properties of the verification properties using this framework that generalize across many components. The first is the property that there are no proof obligations introduced on the implementation of a method by an "affects" clause. The three keywords to describe method effects can be viewed as permissions, the most restrictive (*oblivious to*) giving the operation neither read nor write permissions and the least restrictive (*affects*) giving the operation both read and write permissions on the field(s) in question. Because an "affects" clause places no restrictions on what an implementation may do to the referenced fields, there is no obligation to prove anything about what the method does. On the other hand, both "preserves" and "oblivious to" clauses introduce proof obligations. These clauses restrict an implementation from writing to the concrete fields which fall in the interference fields in question, and so it must be shown that the implementation does not violate these restrictions.

As with other modular verification techniques, this framework provides for 2 sides: the *client* and the *implementation*. Verification from the client's point of view involves guaranteeing that the method calls inside of a cobegin statement are free from entanglement and cannot have their preconditions violated. Showing this requires an inspection of and reasoning about the interference contract. Specifically, it requires showing that the operations inside of the cobegin statement do not interfere with one another; that is, a field which is preserved or marked oblivious by one operation may not be affected by another operation wihtin the same cobegin statement.

6.2 Modular Verification of Client Code

Unconditional Effects Clauses To verify client code involving a cobegin statement, it must first be shown that there is no interference field that is affected as part of an argument of one operation in the cobegin statement is affected *or preserved* as part of either another argument to that operation or as part of an argument to another operation in the cobegin statement. This is simple when there are no arguments that are repeated in any operation in the cobegin statement (it is trivially true in such a case because of the *clean semantics* of RESOLVE).

```
operation f(x: T1, y: T2)
  affects x@a
  oblivious to x@b
  preserves y
```

```
opeartion g(x: T1, y: T2)
  oblivious to x@a
  affects x@b
  preserves y

cobegin
  f(x, yf)
  g(x, yg)
end
```

Listing 1.8. A simple concurrent program using our syntax

In the Listing 1.8, there are two operations, f and g. From the interference contract, it can be seen that operation f *affects x@a* while operation g is *oblivious to x@a*. Furthermore, operation g *affects x@b* while operation f is *oblivious to x@b*. Because the default mode is *oblivious to*, it is immaterial if there are more fields which are part of T1's interference contract. Therefore, the repetition of argument x in the two calls to f and g does not result in interference, so this cobegin statement is safe.

Conditional Effects Clauses Consider the following code segment, where q is a variable of type BoundedQueue **of** Item which respects interference contract #2 (that is, the one with the most avaialble parallelism) and x and y are variables of type Item.

```
procedure Enqueue (clears e: Item, updates q: Queue)
  affects q@b
  when |q| = 0
    affects q@a
  --when |q| > 0
    --oblivious to q@b

procedure Dequeue (replaces r: Item, updates q: Queue)
  affects q@a

-- assume |q| > 0
cobegin
  Enqueue(q, x)
  Dequeue(q, y)
end
```

Listing 1.9. A simple program using the Queue discussed above

Because the preconditions for all methods in a cobegin block must be met be met before it is considered valid[4], any cobegin statement containing a call

[4] Any cobegin statement has an *implicit precondition* which is the set of states which would result in a valid execution of the cobegin statement if it were a sequential

to Dequeue must be entered with $|q| > 0$. Given this information, it can be concluded that the call to Enqueue is, in fact *oblivious to q@a* in this case and thus does not interfere with the call to Dequeue.

6.3 Modular Verification of an Implementation

In order to map from concrete implementation state to abstract specification state, realizations provide a representation invariant (convention) and a correspondence function (or, more generally, a relation). Our approach for establishing operation independence is to augment this correspondence relation with a partitioning of the constituent concrete state space. That is, an implementation must provide a mapping from the concrete data structure involved in the implementation (contents, preFront, and length) to the partitioned mathematical model of the queue in the parallel refinement. Specifically, it must include a correspondence to q@a and to q@b.

Based on the augmented correspondence information, obliviousness needs to be proved for the code of each operation, under the specified conditions (e.g., nonempty queue for Enqueue). In order for an operation's implementation to meet the obliviousness requirement, all statements in its code must be oblivious to the corresponding parts of the data structure. When a statement does not mention a part of the data structure (e.g., prefront), it is trivially oblivious to that variable. (This observation also requires clean semantics.) Otherwise, a statement may use parts of the data structure from their obliviousness requirement only in operations which, themselves, are oblivious on the corresponding parts of the data structure. The underlying data structure itself might be built from other data abstractions. This is not a problem, because the lack of entanglement of one component can be layered on top of appropriately disentangled realization components.

Finally, an implementation of a method which affects a particular interference field carries no proof obligations—*affects* does not require that a method *actually change* a variable's value, only that it *is allowed to*.

Layering Components An observant reader will notice an interesting feature of the various implementations of BoundedQueue discussed above: they rely on another component, and in particular, a specific interference specification which belongs to that component. This suggests that the intuitive notion that the interference specification somehow "sits between" the abstract specification and implementation is not quite correct?the interference specification must be exposed to clients of a component. With this idea in mind, we can view interference specifications as "plugins" to the abstract specification. An implementation may *respect* zero or more interference specifications, not only zero or one. Furthermore, an interference specification may *refine* another interference specification.

block *given any ordering of its constituent operation and function calls*. Note that this is a *stronger* precondition than the conjunction of the consituent preconditions.

In the case of the BoundedQueue example, the Array's interference specification is important because without it, the underlying Array used in the implementations would appear to those implementations as an atomic entity, distinct elements of which are unable to be concurrently accessed. Obviously, enabling concurrent Enqueue/Dequeue operations on an Array requires concurrent access to distinct elements in the Array. An examination of the IndependentSlots interference specification for Array reveals that concurrent read *or write* accesses to distinct elements is allowed by the specification. This paves the way for our BoundedQueue implementation to execute Enqueue and Dequeue in parallel.

The verification that the implementations of Enqueue and Dequeue in implementation #2 respect their respective interference specifications is relatively straightforward. The interference correspondence provides that the sub-field or q.contents@c at index q.head is in field q@a, and the rest of the sub-fields in q.contents@c are in field q@b. This implementation of Dequeue (reproduced below along with Enqueue) only accesses the item at index q.head of q.contents (which implies that it only affects the sub-field q.contents@c at index q.head).

This implementation of Enqueue accesses only the item of q.contents at index q.postTail (which implies that it only affects sub-field q.contents@c at index q.postTail). When q = 0, this method body is allowed to affect q@a (so it can affect q.contents@c at index q.head) and thus carries no proof obligations. When |q| > 0, it must be shown that the call to SwapItem on line 2 does not affect q.contents@c at index q.head (that is, it must be shown that q.postTail /= q .head). From the correspondence, it can be deduced that q.head = q.postTail ==> |q| = 0 and therefore |q| > 0 ==> q.head /= q.postTail, so the call on line 2 does not affect the restricted parts of the data structure.

```
procedure Enqueue (clears e: Item, updates q: Queue)
  SwapItem(q.contents, q.postTail, e)
  Clear(e)
  Increment(q.postTail)
5 Mod(q.postTail, maxLength)
end Enqueue

procedure Dequeue (replaces r: Item, updates q: Queue)
  SwapItem(q.contents, q.head, r)
10 Increment(q.head)
  Mod(q.head, maxLength)
end Dequeue
```

7 Summary and Future Directions

This paper has presented a novel framework for modular verification of concurrent programs using data abstractions. Specifically, it has explained how multiple operations can be simultaneously invoked on an abstract data object if a set of interference conditions can be specified and verified using an augmentation to

the abstract specification of the data abstraction. The proof process is strictly modularized. The paper has presented a concrete example to illustrate the ideas. Future directions include development of a formal proof system and automated verification.

Acknowledgments This research is funded in part by 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. M. Bagherzadeh and H. Rajan. Panini: A concurrent programming model for solving pervasive and oblivious interference. In *Proceedings of the 14th International Conference on Modularity*, MODULARITY 2015, pages 93–108, New York, NY, USA, 2015. ACM.
2. R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 97–116, New York, NY, USA, 2009. ACM.
3. D. E. Harms and B. W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Trans. Softw. Eng.*, 17(5):424–435, May 1991.
4. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
5. G. Kulczycki. *Direct Reasoning.* Phd dissertation, School of Computing, 2004.
6. G. Kulczycki, M. Sitaraman, W. F. Ogden, and B. W. Weide. Clean semantics for calls with repeated arguments. Technical Report RSRG-05-01, Clemson University - School of Computing, March 2005.
7. K. R. M. Leino, P. Müller, and J. Smans. *Foundations of Security Analysis and Design V: FOSAD 2007/2008/2009 Tutorial Lectures*, chapter Verification of Concurrent Programs with Chalice, pages 195–222. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
8. W. Mostowski. *Verified Software: Theories, Tools, and Experiments: 7th International Conference, VSTTE 2015, San Francisco, CA, USA, July 18-19, 2015. Revised Selected Papers*, chapter Dynamic Frames Based Verification Method for Concurrent Java Programs, pages 124–141. Springer International Publishing, 2016.
9. E. Rodrguez, M. Dwyer, C. Flanagan, J. Hatcliff, and G. T. Leavens. Extending JML for modular specification and verification of multi-threaded programs. In *In ECOOP, LNCS 3586*, pages 551–576. Springer, 2005.
10. M. Sitaraman, B. Adcock, J. Avigad, D. Bronish, P. Bucci, D. Frazier, H. M. Friedman, H. Harton, W. Heym, J. Kirschenbaum, J. Krone, H. Smith, and B. W. Weide. Building a push-button resolve verifier: Progress and challenges. *Formal Aspects of Computing*, 23(5):607–626, 2011.

11. J. Smans, B. Jacobs, and F. Piessens. Vericool: An automatic verifier for a concurrent object-oriented language. In *Proceedings of the 10th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems*, FMOODS '08, pages 220–239, Berlin, Heidelberg, 2008. Springer-Verlag.
12. S. T. Taft. Multicore programming in parasail. In *International Conference on Reliable Software Technologies*, pages 196–200. Springer, 2011.