Enabling Modular Verification with Abstract Interference Specifications for a Concurrent Queue

Alan Weide, Paolo A. G. Sililotti, and Murali Sitaraman

Technical Report RSRG-16-03
School of Computing
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

May 2016

Enabling Modular Verification with Abstract Interference Specifications for a Concurrent Queue

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 the different threads, and (ii) the lack of abstraction in the description of read/write effects of operations on shared data structures. While the approach proposed in the paper is intended to address both of these challenges, the focus of this paper is on the second challenge. The paper describes a specification and verification framework in which abstract specifications of strictly 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-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. While both the problem and the solution are of interest, this paper, focuses only on a solution to the second problem.

To illustrate the ideas, the paper presents a bounded queue data abstraction and outlines three different implementations 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 that might introduce additional modeling or interference or other details 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. 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 abstract and interference specification is done once in the lifetime of the implementation. Verification of client code relies strictly on the specifications.

This paper is strictly work in progress. We outline, for example, the specification and verification framework, but do not include formal proof rules. The rest of the paper is organized as follows. Section 2 summarizes the most related work. Section 3 describes the central example and alternative implementations. Section 4 describes the solution. It begins with a presentation of the interference specification that forms the basis for the subsequent discussion on verification. The last section summarizes and gives directions for further research.

2 Related Work

The summary here is meant to be illustrative of the type of related work, not exhaustive.

Classical solutions to the interference problem (e.g., [3]) would involve defining and using locks, but neither the solutions nor the proofs of absence of interference here involve abstraction or specification. Solutions based on CAS 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 in [7] 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 [6] address safe concurrent execution in the context of more general solutions to address aliasing and

sharing for automated verification. The work in [9] 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 [5]. Using them, it is possible to estimate an upper bound on the location sets that may be affected by a thread in Chalice.

3 Bounded Queue Data Abstraction Specification and Alternative Implementations

3.1 Abstract Specification

Here we present a RESOLVE-style specification for a bounded queue [8]. BoundedQueueTemplate is implemented in three different ways, each of which exhibits different opportunities for parallelization, and which are described in detail in the next section. A specification for these parallelization opportunities is proposed in a later section.

Abstractly a queue is modeled as a mathematical string of items in BoundedQueueTemplate. The concept defines operations, such as Enqueue, Dequeue, SwapFirstEntry, Length, and RemCapacity. The operations have been designed and specified to avoid aliasing that arises when queues contain non-trivial objects [2] and to facilitate clean semantics [4]. In addition to the operations, a swap operator is defined on all types to facilitate data exchange without deep or shallow copying [2]. Since this paper is concerned mainly with latter challenge of concurrent execution of data abstraction operations, we omit further discussion of this idea, though it is critical.

operation Enqueue (**alters** e: Item, **updates** q: Queue)
  requires |q| < MAX_LENGTH
  ensures q = #q o <#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> o 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> = substring(#q, 0, 1) **and**
    q = <#e> o substring(#q, 1, |#q|)

SwapFirstEntry operation makes it possible to retrieve or update the first entry, without introducing aliasing. Here, substring is a mathematical notation.

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

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

The function operations Length and RemCapacity behave like one would expect: Length returns an integer equal to the number of elements in the queue, and RemCapacity returns an integer equal to the number of free slots left in the queue before it becomes full.

3.2 Alternative Implementations

We have developed three alternative circular array implementations of the bounded queue specified above, each with different parallelization opportunities. In the first two implementations, the length of the underlying array is equal to the maximum length of the queue, MAX_LENGTH, and in the third the length of the array is MAX_LENGTH + 1.

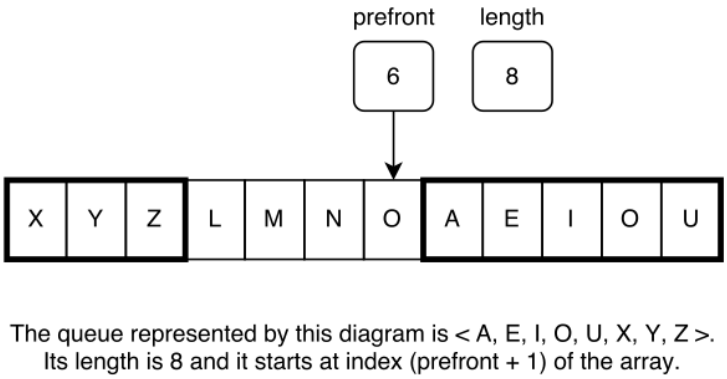

Fig. 1. An implementation of a bounded queue using two Integer fields, prefront and length

The first implementation has two fields: two Integers prefront and length. prefront is the index of the array just before the first element of the queue, and length is the number of elements in the queue. This implementation cannot handle concurrent calls to Enqueue and Dequeue without synchronization because both of those calls must necessarily write to length. A client can, however, make concurrent calls to SwapFirstEntry and Enqueue when the precondition

for both methods is met before the parallel block (that is, if 0 < |q| and |q| < MAX_LENGTH. These two methods may be executed in parallel because SwapFirstEntry touches only the head of the queue and does not modify length, while Enqueue will write length and touch the end of the queue (which we know is different from the head of the queue because there was already an element in the queue before Enqueue was called). An empty queue in this implementation has length = 0 **and** 0 <= prefront < MAX_LENGTH, and a full queue has length = MAX_LENGTH **and** 0 <= prefront < MAX_LENGTH.

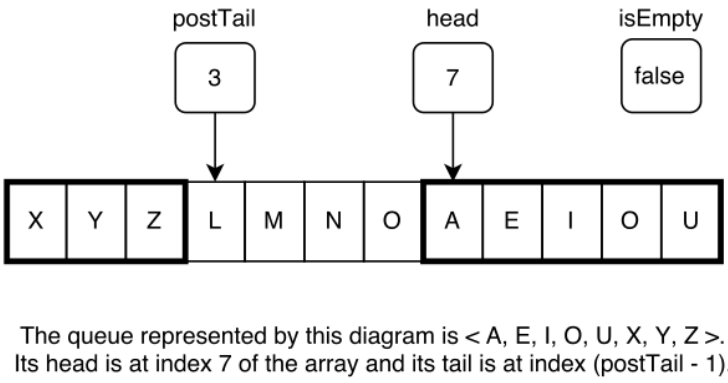

Fig. 2. An implementation of a bounded queue using two Integer fields, head and postTail, and a Boolean field isEmpty

The second implementation also has two Integer fields: head and postTail, and an additional Boolean field isEmpty. head is the index of the array at which the first element of the queue is located, and postTail is the index of the first element of the array after the last element of the queue. isEmpty is necessary to distinguish between a full queue and an empty queue (in both cases, head = postTail). As in implementation 1, a client can concurrently call Enqueue and SwapFirstEntry as long as both preconditions are satisfied. However, because the length of the queue is computed from the head and postTail fields (and not another variable written by both Enqueue and Dequeue), we can also concurrently call Enqueue and Dequeue, but only in a more limited set of circumstances than is described by their respective preconditions: the queue must have at least 2 entries in it and there must be at least 2 "free" slots in the array. This restriction is important because both Enqueue and Dequeue must at least read isEmpty to determine if the queue is empty when head = postTail. By restricting concurrent calls to these two methods to those situations when isEmpty will not be changed by either method (that is, when the queue will be made neither full nor empty by either Enqueue or Dequeue), we can guarantee deterministic behavior when they are executed in parallel.

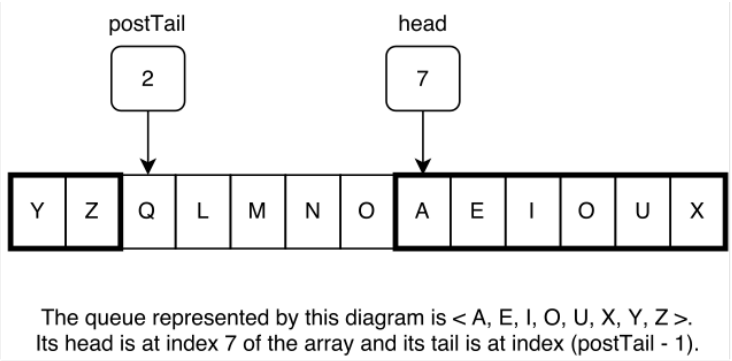

Fig. 3. An implementation of a bounded queue using two Integer fields, head and postTail, and a sentinel node to distinguish between a full queue and an empty one

The third and final implementation is similar to the second in that its two Integer fields are head and postTail (and they represent the same things), but in lieu of a Boolean isEmpty field, there is a sentinel node added to the array so that when head = postTail it can only be the case that the queue is empty (a full queue has head = (postTail + 1) **mod** (MAX_LENGTH + 1)). Because the length of the array is greater than MAX_LENGTH, there will always be some element of the array that is not part of the queue. This differentiation between a full and empty queue without the need to have a separate variable ensures that even when the queue might become either full or empty during a call to Enqueue or Dequeue, it will not write anything that the other method reads or writes.

4 Interference Specification and Modular Verification

Modular reasoning about the safe execution of concurrent threads can be separated into three distinct parts: (i) a description of the conditions under which operations are independent, (ii) a proof that client code ensures these independence conditions, and (iii) a proof that an implementation guarantees non-interference under these conditions.

Our approach to these three tasks is described below and illustrated using the first bounded queue realization from the previous section.

4.1 Interference Specification

A functional specification, as given in section 3.1, does not reveal the degree to which different parts of the abstract state are entangled in the implementation. The correspondence relation between concrete state and abstract state is part of the proof of correctness for the implementation, and modular verification precludes its use in reasoning about client code.

Reasoning about the independence of concurrent threads in client code, however, requires exposing more information. Our approach for the interference specification involves segmenting the abstract state into orthogonal components, and encapsulating the description of this segmentation in a distinct specification, a parallel refinement. While this segmentation is all that is necessary for the interference specification for the example in this paper, in general, the augmentation may additionally supplement the abstract model with more elaboration in order to specify absence of interference among operations. In this case, the specification will also need to state the additional guarantees (ensures clauses) on the supplemental model for each operation, not just interference-related specifications as in the present example.

A parallel refinement for the first implementation of the bounded queue is given below.

```
Parallel refinement Version_1 for Bounded_Queue_Template
  Type Queue is modeled by Str(Entry)
    Exemplar q
    renaming substring(q, 0, 1) as front
    renaming substring(q, 1, |q|) as rest
    Lemma Non_Interference: q = q.front o q.rest

  operation Enqueue (alters e: Item; updates q: Queue)
    when q.front /= empty_string oblivious to q.front
    affects q.rest

  operation Swap_First_Entry  (updates r: Item; updates q: Queue)
    oblivious to q.rest
    affects q.front
end Version_1
```

Names for different components of the abstract state are introduced with the renaming keyword. The fact that these components represent a partitioning of the full abstract state space follows from the non-interference lemma.

Notice that this partitioning of the state space is not the same as establishing the *independence* of these components from the point of view of the correspondence relation. In this example, the independence of the Enqueue operation on front is conditioned by the queue being non-empty. These independence conditions are in addition to the usual preconditions of the corresponding operations from the template specification, so Swap_First_Entry is oblivious to rest when the queue is non-empty.

4.2 Modular Verification of Client Code

In order for a set of statements to be safely executed in parallel, each component of the (abstract) state space can be affected by at most one statement. Furthermore, if any component is affected by some statement, all of the other statements must be oblivious to this component.

For example, with the parallel refinement given above, Swap_First_Entry and Enqueue affect non-overlapping components. Furthermore, each is oblivious to the component affected by the other, assuming the queue is non empty. The following client code illustrates the parallel composition of these operations.

```
Assume 0 < |q| < max_length;
cobegin
  Swap_First_Entry(x, q);
  Enqueue(y, q);
end;
```

First we note that the client code above can be executed concurrently only if there is no aliasing between objects x and y. This isolation is implied if the programming language is defined to have a clean semantics like RESOLVE or through disciplined programming in a language to avoid unintended aliasing. Under clean semantics, the effects of operations are restricted to their explicit parameters (or explicitly specified global variables) [4]. The rest of the discussion assume that the language already includes a solution to the aliasing problem and facilities clean reasoning.

In addition to satisfying the usual preconditions for functional correctness, the verification of the client code includes establishing the independence conditions of the two operations. This verification is carried out entirely in the context of the client code, using only the abstract functional specification and parallel refinement of the queue template.

The independence of the constituent statements of a cobegin block means that the statements can be executed in any concurrent or arbitrarily interleaved manner. The semantics of their execution is identical to that of their sequential composition.

4.3 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 relation, more generally). 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 front and to rest.

Based on the augmented correspondence information, obliviousness needs to be proved for the code of each operation, under the specified conditions (e.g., non-empty 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.

```
Realization CircularArrayRealiz for BddQueueTemplate with parallel
    refinement Version_1;

  Type queue = Record
    contents: array 0..MAX_LENGTH − 1 of item;
    prefront, length: Integer;
  end;
    convention
      0 <= q.prefront < MAX_LENGTH and
      0 <= Q.length <= MAX_LENGTH;
    correspondence
      Conc.q = Iterated_Concatenation(i = q.prefront + 1.. q.prefront + q.
      length, q.contents(i mod MAX_LENGTH));

    correspondence for Version_1
      Conc.q.front = q.contents(q.prefront + 1 mod MAX_LENGTH);
      Conc.q.rest = Iterated_Concatenation(i = q.prefront + 2.. q.prefront + q.
      length, q.contents(i mod MAX_LENGTH));
  end;


  Procedure Enqueue(alters e: item; updates q: queue);
    q.length := q.length + 1;
    q.contents[q.prefront + q.length mod MAX_LENGTH] :=: e;
  end Enqueue;

  Procedure SwapFirstEntry(updates r: item; updates q: queue);
    q.contents[q.prefront + 1 mod MAX_LENGTH] :=: r;
  end SwapFirstEntry;
end CircularArrayRealiz;
```

In the implementation, :=: is the swap operator. The proof of obliviousness is seen as follows. Neither Enqueue nor SwapFirstEntry affect q.prefront (which is used in the correspondences of both front and rest.) For Enqueue, when the queue is not empty, q.length >= 1 before it is called. So q.contents that is modified is at least 2 away from q.prefront. Therefore, Enqueue is oblivious to q .contents[q.prefront + 1 **mod** MAX_LENGTH]. In SwapFirstEntry, q.length is not used. Only q.contents[q.prefront + 1 **mod** MAX_LENGTH] is affected, so it is oblivious to the rest of the contents.

5 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. D. E. Harms and B. W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Trans. Software Eng.*, 17:424–435, 1991.
3. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
4. G. W. Kulczycki. *Direct Reasoning.* PhD thesis, Clemson University, Clemson, SC, USA, 2004. AAI3125470.
5. 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.
6. 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, Cham, 2016.
7. 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.
8. 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.
9. 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.