Intermediate Models for Expressing Parallelization Opportunities with Abstract Data Types

Alan Weide (weide.3@osu.edu), The Ohio State University, Columbus, OH 43210

Abstract—Concurrency support is important in a modern programming language, but naïve nondeterministic approaches lead to subtle bugs that are difficult to track down. Thus, there is a need for a model of concurrency that provides safe, guaranteed deterministic behavior. Current deterministic concurrent programming models rely on the concrete representations of objects to express how a component may be safely parallelized, but this flies in the face of the RESOLVE model of computation, where all that should matter are variables’ abstract mathematical-model values. A preliminary approach to using an “intermediate model” is proposed to begin the inclusion of safe, verifiable, deterministic concurrency in the RESOLVE programming language. A bounded queue serves as an example.

Index Terms—RESOLVE, concurrency, determinism, parallelization, queue

I. INTRODUCTION

When abstractly expressing how a component may be safely parallelized, the usual mathematical model is often not expressive enough to support both a description of allowed concurrency and automatic verification. One operation may only change a small part of the mathematical model while another operation changes a different part. In the case of a bounded queue, the obvious parallelization is to allow concurrent enqueue and dequeue operations. It’s possible to express this potential for concurrency in an abstract way in the basic model of a bounded queue (that is, a string of Items) using substrings, but it becomes nearly impossible to devise proof rules using that expression because to do so, parts of the underlying data structure must be inferred to correspond to particular substrings of the model, and these correspondences may not always exist. To remedy these problems, this paper introduces the concept of an “intermediate model” in which we split up the original abstract model into finer pieces and leverage a tighter coupling between the intermediate model and the representation to express how operations may be safely parallelized.

II. BOUNDED QUEUE SPECIFICATION

Following is a RESOLVE-style specification for a bounded queue. BoundedQueueTemplate may be implemented in three different ways, each of which exhibits different opportunities for parallelization, to be expressed using the intermediate models introduced here. BoundedQueueTemplate defines three procedures, Enqueue, Dequeue, and SwapFirstEntry, and two functions, Length and RemCapacity. The operations are specified in terms of the string value of the queue (its mathematical model for functional specification purposes).

procedure 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 the incoming value of e at the end of the queue.

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

The contract for Dequeue says several things. The requires clause says that in order to be called, q must not be empty. The ensures clause says that the old value of q is the concatenation of the outgoing value of the string containing e with the outgoing value of q. Effectively, it says that the first element of the incoming value of q is not part of the outgoing value of q and, furthermore, that element is the outgoing value of e.

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

The contract for SwapFirstEntry looks much different from the contract for Dequeue, but the two operations are very similar in their intent. First, the requires clause for SwapFirstEntry is identical to that of Dequeue. The ensures
clause says that the outgoing value of e is the first entry in the incoming value of q, and that the outgoing value of q is the string containing the incoming value of e concatenated with all of the incoming value of q except for its first element.

```
function Length (restores q: Queue) : Integer
ensures
    Length = |q|
```

```
function RemCapacity (restores q: Queue) : Integer
ensures
    RemCapacity = MAX_LENGTH - |q|
```

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

### III. Summary of Current Work

- **Prefront**
- **Length**

![Diagram](image)

The queue represented by this diagram is `<A, E, I, O, U, X, Y, Z>`. Its length is 8 and it starts at index (prefront + 1) of the array.

![Fig. 1—A representation of BoundedQueueTemplate using two Integer fields, prefront and length](image)

Preliminary work has produced a possible intermediate model to express one possible parallelization opportunity based on a representation of the bounded queue in Fig. 1. This implementation can support concurrent invocations of Enqueue and SwapFirstEntry as long as both requires clauses are satisfied before the parallel block. The proposed intermediate model (from [3]) specifies the queue as follows:

- **parallel refinement** Version1 for BoundedQueueTemplate
- **type** Queue is modeled by string of Item
- **exemplar** q
  - renaming substring(q, 0, 1) as front
  - renaming substring(q, 1, |q|) as rest
- **lemma** NonInterference: q = q.front * q.rest
- **operation** Enqueue (alters e: Item, updates q: Queue)
  - when q.front /= empty_string
  - oblivious to q.front
  - affects q.rest
- **operation** SwapFirstEntry (updates r: Item, updates q: Queue)
  - oblivious to q.rest
  - affects q.front

```
end Version1
```

This bit of code says several important things. First, it says that this intermediate model, called “Version1”, is a parallel refinement for BoundedQueueTemplate, which is the contract for the bounded queue specified above. Next, it renames parts of the string of Item that models the queue as “front” and “rest”, which is useful in describing the effects of the methods. The lemma states how the intermediate model is related to the original abstract model the abstract value of this queue is the concatenation of q.front and q.rest.

In addition to a slicing-up of the top-level model, a parallel refinement or intermediate model includes augmented contracts for methods that may be safely executed in parallel under certain circumstances. The augmented contract for Enqueue states that when q.front /= empty_string (that is, exactly when q /= empty_string), the Enqueue operation is “oblivious to” q.front, meaning it neither reads nor writes any part of the representation that corresponds to q.front. The affects clause states that Enqueue always touches q.rest, and may modify the part of the representation that corresponds to it. Analogously, the contract for SwapFirstEntry states that the method is always oblivious to q.rest, which implies that any method which affects q.rest may safely be called concurrently with SwapFirstEntry (assuming, of course, that there are no other interfering effects).

Note the absence of an augmented contract for the Dequeue operation. Dequeue can never be safely executed in parallel with any other operation in this refinement of BoundedQueueTemplate because it necessarily affects q.front and q.rest.

### IV. Conclusion

There has been a substantial amount of progress made by others in developing specifications for how data structures interact in a concurrent environment, but it remains largely relegated to the concrete domain, limiting its utility in verified software that relies on abstraction. We have proposed an intermediate model aimed at abstracting the parallelization potential of software to a level above the concrete representation so that concurrent programs may be verified to be correct and deterministic, and we have demonstrated its potential utility in a simple bounded queue component.

### Acknowledgment

Special thanks to Paolo A. G. Sivilotti and Murali Sitaraman for their contributions to the core of this work.

### References