Design and Specification of Iterators Using the Swapping Paradigm

Bruce W. Weide, *Member, IEEE*, Stephen H. Edwards, Douglas E. Harms, *Member, IEEE*, and David Alex Lamb, *Senior Member, IEEE*

***Abstract*—How should iterators be abstracted and encapsulated in modern imperative languages? We consider the combined impact of several factors on this question: the need for a common interface model for user defined iterator abstractions, the importance of formal methods in specifying such a model, and problems involved in modular correctness proofs of iterator implementations and clients. A series of iterator designs illustrates the advantages of the swapping paradigm over the traditional copying paradigm. Specifically, swapping based designs admit more efficient implementations while offering relatively straightforward formal specifications and the potential for modular reasoning about program behavior. The final proposed design schema is a common interface model for an iterator for any generic collection.**

***Index Terms*—Common interface model, formal specification, iterator, modular reasoning, program verification, proof of correctness, swapping**

I. Introduction

An *iterator* is an abstraction that supports sequential access to the individual items of a collection, without modifying the collection. Although some "academic" languages (most notably Alphard [16] and CLU [13]) include special language constructs for iterators, and others have been proposed [3], the most widely used modern imperative languages, such as Ada and C++, offer no special support for iterators. In these languages, iterators must be designed and encapsulated using the same mechanisms that are used for other user-defined abstractions: types, procedures, and packages/classes/modules. This paper discusses why previously published iterator designs are unsatisfactory in several respects, and considers the combined impact of several recent advances on the potential for improvement.

One such development is the proposal by Harms and Weide [9], [19] that swapping should replace copying as the primary data movement mechanism in imperative programs. In the *swapping style* of programming, the usual assignment operator, :=, disappears. (Of course, copying still can be achieved by calling a procedure to do it.) The universal method of data movement becomes the swap operator :=:, which exchanges the values of its two operands. This subtle change leads to several advantages for designing and implementing generic reusable software components, including improved efficiency and simplified modular reasoning about program behavior. The swapping paradigm is especially valuable when dealing with potentially large and complex data structures that represent collections of items—just the situation in which iterators are normally used.

In other recent work, Edwards [4] proposes that the swapping paradigm might be applied to the design and implementation of iterators. He also addresses a serious problem facing software component designers, i.e., developing interface models that simplify component composition. Tracz [18] discusses an example involving what Edwards [5], [6] notices is an iterator. Edwards defines a *common interface model* informally (see [7] for a formal treatment) as a convention, shared by designers of piece-part families and their potential clients, for how the plugs and sockets of plug-compatible software components are supposed to work. It includes not only parameter profiles of operations but also a shared understanding of the abstract behavior of those operations.

A third recent development is the development of formal trace specifications for iterators by Lamb [12] and by Pearce and Lamb [15]. These papers clearly explain the need for, and difficulties in, formal specification of iterators. Two related aspects of this issue that must be faced when defining a common interface model are, How should the abstract behavior of an iterator be designed so that all relevant features can be formally specified, and how can we use this specification to reason about program behavior? Especially in a component-based system, this reasoning must be modular; i.e., it must be possible to reason about the correctness of the iterator implementation independently of each client program, and vice versa. The crucial importance of, and difficulties with, modular verification of realistically large software systems in modern languages with data abstraction are noted by Ernst *et al.* [8] and Hollingsworth [10], among others.

Previous work on iteration over the elements of a composite data structure, summarized nicely by Bishop [1], has not considered together efficiency with respect to copying, the need for formal specification of a common interface model,

Manuscript received August 12, 1992; revised March 1994. The work of the first two authors was supported by the National Science Foundation under Grants CCR-9111892 and CCR-9311702, and by the U.S. Department of Defense Advanced Research Projects Agency (ARPA) under ARPA Contract F30602-93-C-0243, monitored by the U.S. Air Force Material Command, Rome Laboratories, ARPA Order A714. Recommended by M. Moriconi.

B. W. Weide and S. H. Edwards are with the Department of Computer and Information Science, Ohio State University, Columbus, OH 43210 USA; e-mail: weide@cis.ohio-state.edu, edwards@cis.ohio-state.edu.

D. E. Harms is with the Department of Mathematics and Computer Science, Muskingum College, New Concord, OH 43762 USA; e-mail: harms @muskingum.edu.

D. A. Lamb is with the Department of Computing and Information Science, Queen's University, Kingston, ON K7L 3N6 Canada; e-mail: dalamb@qucis.queensu.ca.

IEEE Log Number 9403568.

and the importance of modular reasoning about correctness in the design of iterators. This paper therefore has the following related objectives:

1) To show how to design an iterator in the swapping paradigm, which permits a most-efficient implementation, i.e., one that does not copy either items of the collection or the collection's representation data structure;
2) To give an abstract model-oriented specification of an iterator for a particular abstract collection of items, so the iterator's abstract interface is clearly and unambiguously defined;
3) To explain how this specification supports modular reasoning about the behavior of the iterator implementation and its clients, and modular verification of programs involving iterators; and
4) To demonstrate how the design can be generalized to lead to similar iterators for any abstract collection, thereby promoting composability of components.

This paper is, in effect, a proposal for a common interface model for a large class of iterators. A superficial examination of this model suggests that it is not much different from previously published iterators. In fact, however, our designs resemble others, primarily in having similar names for the operations. The *behavior* of these operations—both in functionality and performance—is subtly but importantly different.

Section II begins with a review of past work on iterators and notes the problems with previous designs. We also review the swapping paradigm and the RESOLVE notation for formal specification, and introduce a simple example that forms the basis for development of a simple iterator: a first-in, first-out (FIFO) queue abstract data type (ADT). Section III explains, step-by-step, how to arrive at the design of an acceptable swapping-style iterator for this ADT. It addresses objectives 1)–3) above for each candidate design along the way. Finally, Section IV discusses variations and extensions, and shows how the method used for the simple FIFO queue example can be generalized to a schema for specifying iterators for arbitrary collection types. All iterators designed using these principles share a common interface model, which can serve as the basis for interfaces exported by Ada generic packages and C++ class templates, among others. Example code for two typical client operations is provided in the Appendix.

II. Background

This section discusses the features required of an acceptable iterator design, the rationale for limiting the discussion to user-defined abstractions (as opposed to built-in language constructs that support iterators), relevant details of the swapping paradigm, and our approach to, and notation for, formal specification. Throughout the discussion, we refer to the client (respectively, "client program" or "client code") and to the implementer (respectively, "implementation"). The former is the programmer (respectively, program) that uses the abstract iterator concept. The latter is the programmer (respectively, program) that realizes the iterator abstraction in the form of an executable code.

A. Iterators

The simplest kind of iterator permits a client program to examine (i.e., to execute some piece of code for) each of the items of a collection without modifying the collection as a side effect of iterating over it. The items are presented to the client in some order that is based on the collection abstraction. Examples include enumerating and accumulating information about the items in a set, printing all the items in a tree, and copying a FIFO queue. There is no natural order for iterating over the elements of a set (any order will do), but there are several useful presentation orders for trees and an obvious natural order for a FIFO queue.

There are various more complex iterators and possible uses for them. For example, we might wish to be able to exit early from an iteration based on satisfaction of some condition, to have some control over the order of iteration or to leave it entirely unspecified and up to the implementer's discretion; or we might wish to change the original collection or its items while iterating over it. We begin by considering the simplest case described above, and discuss more complex cases in Section IV. A review of past work suggests that there are two subtle aspects of even the simplest iterators.

1) *Correctness:* It should not be permissible for a (correct) client program to iterate over a collection while interleaved operations on that collection might be changing it. We call this property *noninterference.*
2) *Efficiency:* It should be possible for a client program to iterate over a collection without copying the data structure that represents the collection and without copying the individual items in the collection.[1]

Correctness: Recognition of the relationship between noninterference and the modular verification of correctness dates back to attempts to verify Alphard programs involving iterators [16]. Programmers using one of Alphard's iterator constructs are advised to consider noninterference to be a restriction on its use, but no formal proof obligation is raised during verification. Proof rules should permit local verification of an implementation and its client programs, but this cannot be achieved without an assurance of noninterference, either through restriction by language syntax or by the presence of a noninterference proof obligation. Alphard, like other languages with iterator constructs, offers neither.

In an attempt to deal with noninterference in user-defined iterator abstractions, Booch [2] and Bishop [1] suggest classifying iterators into two categories, which Booch calls active and passive. An active iterator is a *module* that exports an iterator type and associated operations and permits a client to build iteration loops with standard control constructs, e.g., while loops. The main difficulty with this approach is that such a loop body may also contain calls to operations that manipulate the collection over which iteration is being done; this is precisely the problem with Alphard's and other language-supplied iterator constructs. By contrast, a passive iterator effectively encapsulates the iteration loop in a single

[1] In the special case that copying a collection is the purpose of iterating over it, all copying should take place in the client code that is executed for each item. Copying should not be inherent in the iterator itself.

procedure, which is parameterized by an action that would be the loop body in an iteration using an active iterator. The argument is that in this case there is no (obvious) way for a client to interleave operations that change the collection with those iterating over it, because the latter are encapsulated in the passive iterator procedure.

Unfortunately, passive iterators suffer from their own serious problems, discussed in detail by many authors [1], [2], [4]. From the standpoint of reusability, they are far less flexible than active iterators. For example, a client can iterate simultaneously over multiple collections with an active iterator (see the appendix), but not with a passive one. In the face of formal specification and the need for modular verification, the nature of the action procedure's effects and side effects must be formally specified and proofs modularized. It is not clear how to do this. Moreover, a client still can violate noninterference by, for instance, declaring a collection to be global to the iterator's action procedure and interfering with the iteration by manipulating that collection surreptitiously. The *coup de grâce* for passive iterators from the standpoint of reuse is the observation that an implementation of a passive iterator can be layered easily on top of an active one, but not vice versa.

Therefore, we follow the above-cited papers in concentrating on designs for active iterators. However, we insist that clients observe the noninterference property and be modularly verifiable, which necessarily makes our designs different from previous ones. That is, like Lamb [12], we write our formal specification so that noninterference *must* be observed by a correct client program. A proof obligation involving noninterference is raised in the client that can and must be discharged in a provably correct client program.

By contrast, Booch [2] points out that his iterator designs are relatively unprotected from client abuse. Indeed, nothing but self-discipline prevents a client from altering a collection during iteration over it. The same is true for Bishop's designs [1]. Several methods for repairing this shortcoming are proposed by Edwards [4]; but like Booch and Bishop, he does not deal explicitly with formal specification or the need for a framework for modular verification. These objectives drive many of our design decisions and account for the differences between Edwards's designs and the ones we propose here.

Efficiency: Although noninterference has long been seen as a problem with iterators, Edwards [4] was the first to recognize the inefficiency inherent in both published iterator abstractions and language constructs. All previously published designs for iterators (i.e., those before Edwards's papers [4]–[6]) include a function called, e.g., Value_Of. This returns to the client a *copy* of the next item from the collection.

The execution-time cost of such copying is troubling if the representations of the items in the collection are themselves large, complex data structures. As noted by Harms and Weide [9], the typical method of avoiding this expense—copying only a reference (pointer) to an item, as with the designs recommended by Booch [2] and Bishop [1]—creates even more serious problems from the standpoint of our objectives. It significantly complicates formal specification and, practically speaking, thwarts modular verification [8], [10]. This formal-proof difficulty has practical consequences: It means that human understanding of, and informal reasoning about, program behavior is much harder than it should be. Replacing copying by swapping is both efficient and amenable to tractable formal specification and modular proof rules, and hence to easier understanding of program behavior. This is the reason why we prefer the swapping paradigm for our designs.

Another efficiency issue is noted by Edwards [4] and by Lamb [12]. Achieving optimum performance of an iterator generally requires that the implementer of an iterator have access to the underlying representation of the collection. However, this is not essential solely to obtain the required functionality of an iterator, if the operations on the collection abstraction are sufficiently powerful [4], [9], [19].

B. Language Features and User-Defined Iterator Abstractions

Alphard [16] and CLU [13] have built-in iterator constructs, and Cameron [3] proposes some elegant variations. Here we concentrate on designing iterators as user-defined abstractions in languages that do not include special constructs to support iterators, and we do not further consider possible language support for our designs. There are three reasons for this. First, the practical successors to Alphard and CLU (e.g., Ada and C++) simply do not support iterators directly, so there is clearly a need for a design approach that does not rely on special language support. Second, even with language support, one needs to define formally a common interface model for iterators if a high degree of composability of software components is to be expected [5], [6]. Finally, none of the proposed language mechanisms satisfactorily addresses the problem of noninterference and the need for modular reasoning about program behavior, or the inefficiency of copying.

C. The Swapping Paradigm

The swapping style of software design [9], [19] differs from the conventional copying style in using swapping (and the swap operator :=:) to replace copying (and the standard := operator). It is based on two observations about generic modules, e.g., Ada generic packages. First, items whose types are parameters to generic modules might have large data structures as their concrete representations. These items therefore might be expensive to copy. Second, an attempt to overcome the cost of copying the abstract values of such items by copying references to them inevitably leads to difficulties in establishing program correctness by modular reasoning. This in turn frustrates both the clients of an abstraction and maintainers of its implementations. Therefore, it is advantageous to design the abstract interface of a generic component so that an implementation can achieve data movement by swapping (exchanging) the abstract values of any two variables of the same type, rather than by copying abstract values (destroying old values and duplicating new ones) or by copying references to abstract values.

Harms and Weide [9], [19] and Hollingsworth [10] propose detailed principles to help designers create generic reusable software components in the swapping style. For example, consider the operations on collection types such as a Queue of Items. Insertion operations such as Enqueue should permit

```
concept Queue_Template
   context
      parametric  context
         type Item
   interface
      type Queue is modeled by string of math[Item]
         exemplar q
         initialization
            ensures q = empty_string
      operation Enqueue (
            alters q: Queue
            consumes x: Item)
         ensures    q = #q * <#x>
      operation Dequeue (
            alters q: Queue
            produces x: Item)
         requires  q /= empty_string
         ensures    <x> * q = #q
      operation Is_Empty (
            preserves q: Queue): Boolean
         ensures    Is_Empty  iff  q = empty_string
end Queue_Template
```

Fig. 1. FIFO queue specification.

implementations that swap Items into the structure. Inspection or removal operations such as Dequeue should permit implementations that swap Items out.

A particularly instructive example is an Array of Items ADT. The (single) *primary* operation should take an Array, an index, and an Item, and swap the indexed element with that Item. The usual fetch and store become *secondary* operations using this primitive. That is, they can be implemented with an insignificant performance penalty by layering on top of the primary swap-based operation if they are really needed, and in most clients they are not [9], [19].

D. Formal Specification

The main example we use throughout the rest of this paper is a FIFO queue abstraction. The formal specification of the Queue_Template concept in a dialect of RESOLVE [9], [17], [19] is shown in Fig. 1.

A **concept** specifies a generic abstract module consisting of two parts: **context**, which spells out the information needed to complete the specification, and a description of the exported **interface**. The conceptual context of Queue_Template is provided through a generic parameter, an ADT called Item. The concept exports an ADT called Queue and primary operations to Enqueue and Dequeue Items and to test if a Queue Is_Empty. This is a model-based specification in which a Queue is modeled as a mathematical string of (the mathematical model of) Items. String theory notation includes $\langle x \rangle$, where x is an Item, which denotes the string containing the single Item x; and $a * b$, where a and b are strings, which denotes the string obtained by concatenating a and b. Initially, a variable of type Queue is empty; i.e., its model is the empty string, denoted by **empty_string**.

The notation used in **ensures** clauses (postconditions) is that a variable stands for the value of its mathematical model at the conclusion of the operation; the variable prefixed with # (pronounced "old") stands for the value of the variable's mathematical model at the start of that operation. The # prefix is not needed or used in **requires** clauses (preconditions), where all variables denote values at the start of the operation.

The parameters' *modes* are used to simplify specification, and have nothing to do with the mechanism for passing parameters [9]. Mode **alters** means the argument replacing this formal parameter may be changed as a result of the call; how it is changed is stated explicitly in the postcondition, which generally relates that variable's new value to its old value and to the values of other formal parameters. Mode **preserves** means the argument's value at the conclusion of the operation is the same as it is at the start of the operation. For example, in operation Is_Empty, there is no need to say explicitly in the postcondition q = #q. Mode **consumes** means the argument's value is changed to an initial value for its type. For example, consuming a variable of type Queue would make it equal **empty_string**, while consuming an Integer would make it 0 (assuming the initial value for Integers is 0). Finally, mode **produces** means that the argument's value may be changed by the call, but its value at the beginning of the call has no influence on the operation's behavior.

Lamb [12] and Pearce and Lamb [15] use trace specifications for iterators. In this paper, we use model-oriented specifications like the one above. Model-oriented specifications seem well suited to designs based on swapping, have seen relatively widespread use in practice (e.g., Larch and Z), and are rather easily understood, even by those not intimately familiar with the wide variety of formal specification techniques currently in use [17], [20]. They also have been used in proof systems for modular verification of implementations and clients [8].

At the risk of seeming to apologize for writing formal specifications, we note in advance that the formal specification of the final iterator design we propose is not as short or as simple as we might have hoped. We believe this is due to the moderately complex behavior that the specification describes, inherent in iterators, and not to a serious shortcoming with either the specification notation or with our choice of how to specify iterators in that notation. We know of no comparably complex behavior specified in any formal way that does not *look* at least as imposing. The question arises, though, whether real programmers can be expected to understand such a specification, and, if not, what value it has. Others already have answered this somewhat loaded question [17], [20]. But we would add that the importance of programmer understanding of formal specifications only underscores the need for a common interface model for iterators that, once understood after, say, a careful reading of this paper, leads to rapid understanding of an entire class of structurally similar specifications [6]. We also note that even if most client programmers could understand iterators from only derived metaphorical descriptions and examples and could not read the formalism itself, then a formal specification still would serve an important role as the legal contract between implementer and client against which formal verification could be performed by experts or mechanical provers.

III. Development of an Iterator for a Queue

The goal of this section is to develop a design approach that applies to iterators for any type of collection of any type of item. We create an iterator for the generic Queue type of

Section II, then generalize in Section IV. The presentation in this section is incremental. In each step, we present a proposed design of the iterator and sample client code that uses it, then discuss it, critique it, and propose a new design, until the final design achieves the stated objectives. The development proceeds as follows:

- *Design #1:* Attack problem (1) from Section II-A, i.e., noninterference and modular verification of correctness. We define a companion type Iterator for type Queue with operations that support iteration over a Queue. The idea of this step is to make noninterference a nonissue and thereby permit modular correctness proofs. The chief problem with this design is that it is based on the copying paradigm and therefore is inherently inefficient. In fact, Design #1 might look like a straw man to some readers; after all, no one really designs iterators this way. But that is precisely the point: To enforce noninterference and achieve modularity of correctness proofs, designs based on the copying paradigm must sacrifice efficiency. Other real iterator designs attempt to achieve some degree of efficiency *at the expense of* assured noninterference and proof modularity. Design #1 illustrates that the trade-off might be made in the other direction. It also serves as the basis for better designs to follow.
- *Design #2:* Attack problem (2) from Section II-A, i.e., efficiency with respect to copying. We revise Design #1 to use swapping. The purpose of this step is to permit an implementation of an iterator that still demands noninterference and supports modular verification, yet does not need to copy either the data structure that represents the Queue or any of the Items in it. The main problems with Design #2 are that it is cumbersome to write a loop invariant to demonstrate the correctness of a typical client program, and that some swapping-paradigm principles still are not completely observed.
- *Design #3:* Add some abstract state information to the model of the Iterator type to remedy the verification problem above, and change the operations slightly to take advantage of it. The purpose of this step is to facilitate client correctness proofs and to achieve closer adherence to swapping paradigm design principles. This design achieves all the stated objectives. A generalization that handles arbitrary collections and various extensions is presented in Section IV.

A. Design #1

First, we define a companion type Iterator for the type Queue. This new type has its own operations that support iteration over a Queue. Typical client code involves two steps: Transfer the Queue value into an Iterator variable; then iterate over that variable, not over the original Queue.

An appropriate mathematical model of an Iterator is (like a Queue) a string of Items.[2] This string records the order in which the Items are to be processed during iteration. Here we choose this to be the order in which the Items would be Dequeued from the original Queue. Other orderings can be specified easily by changing the postcondition of Start_Iterator, and, for some representations of type Queue, other orderings can be implemented as easily as the natural order. (See also Section IV.) The specification for Design #1 is shown in Fig. 2.

[2]An Iterator is not modeled by a Queue, because in our model-based specification framework, an ADT's model is always a mathematical object, not another program object.

```
concept Queue_Iterator_Template
  context
    global context
      Queue_Template
    parametric context
      type Item
      facility Queue_Facility is
        Queue_Template (Item)
  interface
    type Iterator is modeled by string of math[Item]
      exemplar i
      initialization
        ensures i = empty_string
    operation Start_Iterator (
        produces i: Iterator
        preserves q: Queue)
      ensures   i = q
    operation Finish_Iterator (
        consumes i: Iterator)
    operation Get_Next_Item (
        alters i: Iterator
        produces x: Item)
      requires   i /= empty_string
      ensures    <x> * i = #i
    operation Is_Empty (
        preserves i: Iterator): Boolean
      ensures    Is_Empty  iff  i = empty_string
end Queue_Iterator_Template
```

Fig. 2. Queue_Iterator Design #1.

Discussion: Design #1 involves a specification mechanism called a *facility parameter*. A **facility** is an instance of a (generic) concept. In this case, Queue_Iterator_Template is parameterized by type Item, and by an instance of Queue_Template called Queue_Facility, which exports a Queue (of Items) ADT and associated operations.

As noted earlier, we should be able to *layer* the implementation of an iterator on top of the corresponding collection abstraction, so that the new code respects the collection abstraction, and this could be done here [9], [14], [19]. However, there are potential order-of-magnitude efficiency gains if the underlying collection and the iterator are implemented together as a single program unit with shared knowledge of the collection and iterator representations. We specify such a composite concept in Fig. 3.

Queue_With_Iterator_Template is a concept that exports the combined interfaces of Queue_Template and Queue_Iterator_Template. The **local context** section in Fig. 3 simply ties down the parameters of these two generic abstractions, so the combination of interfaces is what we require from the strong typing standpoint. This is the RESOLVE mechanism for specification or interface inheritance [11]. In subsequent discussions of efficiency of iterator operations, we refer to the direct implementation of Queue_With_Iterator_Template from Fig. 3.

Here is a sample of client code for iteration using Design #1:

```
Start_Iterator (i, q)
```

```
concept Queue_With_Iterator_Template
  context
    global context
      Queue_Template
      Queue_Iterator_Template
    parametric context
      type Item
    local context
      facility Queue_Facility is
        Queue_Template (Item)
      facility Queue_Iterator_Facility is
        Queue_Iterator_Template (Item, Queue_Facility)
  interface
    re-exports
      Queue_Facility
      Queue_Iterator_Facility
end Queue_With_Iterator_Template
```

Fig. 3. Queue_With_Iterator Specification.

```
while not Is_Empty (i) do
   Get_Next_Item (i, x)
   (* code to process x *)
end while
Finish_Iterator (i)
```

It is evident from the sample code that Design #1 achieves noninterference by defining it away. The original Queue q is completely separate from the Iterator i. The Start_Iterator operation protects q from being changed during iteration. If the code in the loop body of the sample code manipulates q, there is no interference with the iteration. Similarly, changes to x in the code to process x do not influence either q or i. Therefore, it is acceptable for a client program to manipulate q inside a loop that is iterating over i, even if i was obtained from q.

Critique: Noninterference is assured here only at the cost of efficiency. Design #1 effectively forces an implementation of Start_Iterator to copy q into i. The reason is that simply copying a reference to q or references to its Items creates aliases, and hence cannot preserve the independence of the abstract values of q and i [9], [10]. It is impossible to prove that such an implementation of Queue_With_Iterator_Template is correct outside the context of a client program, because the client program might manipulate q or its Items through these aliases. The only way to create a modularly verifiable implementation for Design #1 is to copy q (including all of its Items).

However, a clever implementation of Queue_With_Iterator_Template might defer copying the data structure that represents q (but not its Items), as long as there are no calls to Enqueue or Dequeue on the original Queue q during an iteration over i. It can keep enough internal state as part of a Queue representation to recognize that in the abstract view of these operations, q supposedly has been copied into an Iterator i. It can determine whether an iteration is in progress by monitoring whether the call to Start_Iterator has been matched by a bracketing call to Finish_Iterator. If a call to Enqueue or Dequeue occurs during an iteration, the copy of q's data structure can be made at that time. Supporting this kind of implementation is the only real reason for the Finish_Iterator operation in Design #1. In the worst case, though, copying of q is still necessary.

```
concept Queue_Iterator_Template
    conceptual context
        uses
            Queue_Template
        parametric context
            type Item
            facility Queue_Facility is
                Queue_Template (Item)
    interface
        type family Iterator is modeled by (
                future: string of math[Item]
                present: math[Item]
                original: string of math[Item])
            exemplar i
            initialization
                ensures i.future = empty_string  and
                        is_initial (i.present)  and
                        i.original = empty_string
        operation Start_Iterator (
                produces i: Iterator
                consumes q: Queue
                produces x: Item)
            ensures     i.future = #q  and
                        i.present = x  and
                        i.original = #q
        operation Finish_Iterator (
                consumes i: Iterator
                produces q: Queue
                consumes x: Item)
            requires    i.present = x
            ensures     q = #i.original
        operation Get_Next_Item (
                alters i: Iterator
                alters x: Item)
            requires    i.future /= empty_string  and
                        i.present = x
            ensures     <x> * i.future = #i.future  and
                        i.present = x  and
                        i.original = #i.original
        operation Is_Empty (
                preserves i: Iterator): Boolean
            ensures     Is_Empty  iff  i.future = empty_string
end Queue_Iterator_Template
```

Fig. 4. Queue_Iterator Design #2.

We again note that nearly all previously published iterator designs do not force copying of the data structure representing the collection, but they *do* force copying of its Items in the course of iterating. In such designs, the counterpart of Get_Next_Item is a function that returns a copy of the next Item in the collection. Again, a modularly verifiable implementation may not make this copy cheaply by creating an alias to the Item. These problems are intrinsic to the copying paradigm [9], [19].

B. Design #2

Design #1 can be changed to use the swapping paradigm. The reason for doing this is to permit an implementation that does not need to copy either the data structure that represents the Queue or any of the Items in it. Two key ideas make this approach workable.

The first is a change to Start_Iterator and Finish_Iterator. Start_Iterator can be modified so that an implementation can move the original Queue into the Iterator object, and the matching call to Finish_Iterator can move the Queue back. This design relieves the implementer from responsibility for copying the data structure the represents the Queue. Moving arbitrarily large data structures in this way can be accomplished in constant (uniformly bounded) time with swapping [9].

The second idea is to define Get_Next_Item so that its implementation does not need to return a copy of the Item to the client, but can swap it out. This is possible if the client is required to pass that Item back (unchanged) in the *next* call to Get_Next_Item. In this case, the implementation can simply put the Item back into the Queue data structure, and swap out the next one to return to the client. The only real hurdle is to get the boundary conditions correct, so that the first and last calls to Get_Next_Item are not special cases.

The mathematical model of an Iterator becomes an ordered triple: a string of Items (called future) serving the same purpose as the model in Design #1, a single Item (called present) that records the Item value currently held by the client, and a string of Items (called original) that records the value of the original Queue. The complete specification for Design #2 is shown in Fig. 4, where the predicate **is_initial** means that its argument has an initial value for its type.

Discussion: Below is a sample of client code for iteration using Design #2.

```
Start_Iterator (i, q, x)
while not Is_Empty (i) do
   Get_Next_Item(i, x)
   (* code to process x without
     changing i or x *)
end while
Finish_Iterator (i, q, x)
```

Why is this specification so much more complex than Design #1? How does it permit the implementer to avoid copying the Queue data structure and its Items? How can a client check the preconditions of the Get_Next_Item and Finish_Iterator operations? We answer these and other questions below by considering how to implement Queue_With_Iterator_Template.

Fig. 5 traces an example of the effects of the sample client code segment above. It shows both the abstract models of i, q, and x (to illustrate the abstract behavior), and the critical aspects of possible concrete representations for i and q (to support performance claims). In this case, q is a Queue of Integers,[3] mathematically modeled as a string of mathematical integers; strings are shown between $\langle\ \rangle$. Fig. 5 also shows a typical Queue representation, which is a record containing two fields: f points to the front node of the queue and r to the rear. The representation of an Iterator is identical, except that there is an additional field in the representation record: p points to the node whose Item is presently held by the client (if any). These concrete representations are only illustrative; others also would achieve the claimed performance.

In the top row of Fig. 5, just before execution of the sample client code begins, i and x might have any values. For example, i might have an initial value for type Iterator and x might be 17, as illustrated. The value of x before Start_Iterator is immaterial; it is just a priming value, and the specification does not say exactly what Start_Iterator (q, i, x) returns in x. But note that *i.present* records that value; see the second row of Fig. 5. The next three rows show the situation after the three calls to Get_Next_Item that occur in the case that the original q is modeled by the three element string $\langle 9\ 6\ 90\rangle$. The value of x after the call to Finish_Iterator is 0, because the specification says that operation consumes x.

[3]This makes it easy to understand the operation of the iterator, but it also makes the example too simple to illustrate the importance of not copying an Item, which might be a far more complex type than Integer! We opted for ease of understanding in choosing the example.

One aspect of Fig. 5 might seem mysterious: Why are there top-level pointers to the records representing an Iterator and a Queue? These pointers are not strictly necessary in order to achieve the claimed performance; swapping of Iterators and Queues still would require only constant time, even without this extra level of indirection. However, it is important for implementing swapping in a uniformly bounded time, and for code-sharing among instances of generics, as noted in [9].

In the abstract explanation of Start_Iterator, the original value of q is remembered in $i.future$, from which Items subsequently are to be dispensed to the client by Get_Next_Item. An implementation of Start_Iterator in Design #2 need not copy the original Queue data structure in order to achieve this effect. It can acquire the original value of q by swapping. Start_Iterator is designed to consume q in order to support this implementation.

On first reading, it might appear that Start_Iterator should have to copy q in order to satisfy the postcondition clause `i.original = #q`. This also is not the case, because *i.original* is part of the *abstract* state of an Iterator. There is no implication that the *concrete* representation of an Iterator must explicitly include *i.original*, and indeed none of the other operations demands that *i.original* actually be kept for correct execution, as explained below. Adding an adjunct variable (a variable that participates in proofs but not in executable code) to the Iterator representation enables us to write a formal correspondence relation between the representation and abstract values [10].

Similarly, the postcondition clause `i.present = x` in Start_Iterator means that the Item value returned to the client in x is remembered as part of the Iterator's state. But as above, this does not require copying, because *i.present* is only part of the abstract state of an Iterator and need not be represented concretely, unless some operation's implementation calls for that; none does here.

Similarly, Get_Next_Item need not copy an Item. Its precondition `i.present = x` requires that the client pass in as x an Item equal to the one most recently returned by Start_Iterator or Get_Next_Item. The implementation can merely put this value back into the Queue data structure (in the node referenced by field p in Fig. 5) and return the next Item by swapping it out of the structure. Again, there is no need for copying, because the Item returned must be passed back in the next call to Get_Next_Item, and so on.

When iteration is completed, the client calls Finish_Iterator. This operation's precondition requires that the client give back the one outstanding Item (whose value is *i.present*), at which point the implementation has the entire data structure and all the Items in the original Queue. It simply swaps this with parameter q to achieve the stated postcondition.

A point worth noting is that no code in the client or in the implementation *checks* the clause `i.present=x` at the

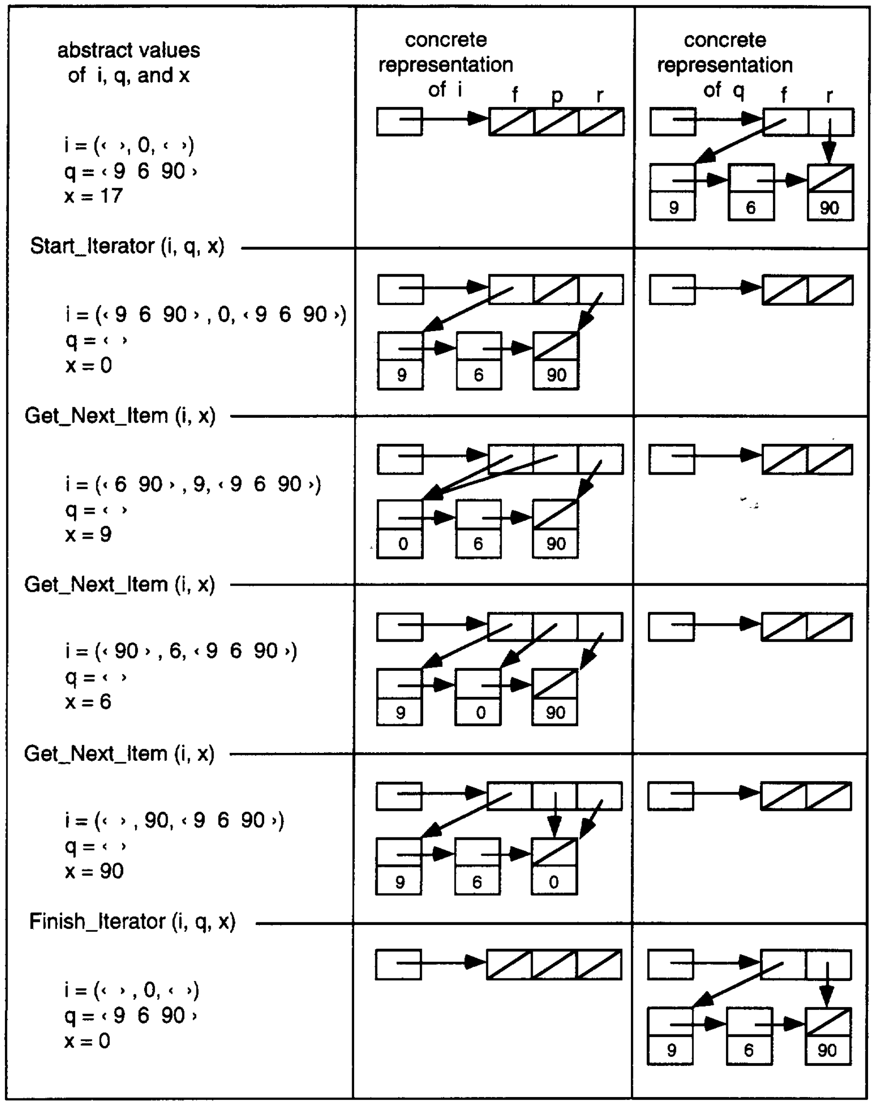


Fig. 5. Sample execution for design #2.

beginning of a call to Get_Next_Item or Finish_Iterator. In fact, because there is no operation that reveals the value of `i.present`, a client or an implementer cannot write such code without copying Items. Thus, the only means for a client to be sure that no preconditions are violated is to be able to *prove* that the code to process x does not change x.

Without the precondition on Get_Next_Item and Finish_Iterator, no such proof obligation would be raised in an arbitrary client program. Although it then might be possible to verify a particular use of the swapping-based implementation, there would be no way to separate a proof of correctness of the implementation from that of the client program. Therefore, we could not prove the correctness of this implementation in a modular fashion, and we could not declare the swapping-based implementation of Queue_Iterator_Template to be correct out of the context of a particular client. The feasibility of such a modular correctness proof was one of the primary objectives of our design.

Critique: Although Design #2 has a more complex specification than Design #1, its swapping-based implementation is straightforward and efficient. However, experience using the specification of Design #2 suggests some minor changes. Most importantly, with Design #2, it is cumbersome to show in the sample client program that the code to process x actually is executed for every item in the original Queue q. The proof relies on a loop invariant that keeps track of the Items that have been processed and relates them to the Items in $i.future$ and the original Queue. It is possible to introduce an adjunct variable for each loop to keep track of the processed Items, but it is more convenient to include support for this in the specification. This and other minor modifications are discussed in the next section.

```
concept Queue_Iterator_Template
    conceptual  context
        uses
            Queue_Template
        parametric  context
            type Item
            facility Queue_Facility is
                Queue_Template (Item)
    interface
        type family Iterator is modeled by (
                past: string of math[Item]
                present: math[Item]
                future: string of math[Item]
                original: string of math[Item]
                deposit: math[Item])
            exemplar i
            initialization
                ensures i.past = empty_string  and
                        is_initial (i.present)  and
                        i.future = empty_string  and
                        i.original = empty_string  and
                        is_initial (i.deposit)
        operation Start_Iterator (
                alters i: Iterator
                consumes q: Queue
                consumes x: Item)
            requires    is_initial (i)
            ensures     i.past = empty_string  and
                        i.present = x  and
                        i.future = #q  and
                        i.original = #q  and
                        i.deposit = #x
        operation Finish_Iterator (
                consumes i: Iterator
                produces q: Queue
                alters x: Item)
            requires    i.present = x
            ensures     q = #i.original  and
                        x = #i.deposit
        operation Get_Next_Item (
                alters i: Iterator
                alters x: Item)
            requires    i.present = x  and
                        i.future /= empty_string
            ensures     i.past = #i.past * <x>  and
                        i.present = x   and
                        <x> * i.future = #i.future  and
                        i.original = #i.original   and
                        i.deposit = #i.deposit
        operation Is_Empty (
                preserves i: Iterator): Boolean
            ensures     Is_Empty  iff  i.future = empty_string
end Queue_Iterator_Template
```

Fig. 6. Queue_Iterator Design #3.

C. Design #3

In Fig. 6, we add to the abstract model a field (called past) that records the Items that have been returned to the client through Get_Next_Item, and a field (called deposit) that records the priming Item that the client passed into the first call to Start_Iterator. We also add a precondition to Start_Iterator to guarantee that the Iterator i satisfies its initial condition, make Start_Iterator consume the deposited value x, modify the post condition of Finish_Iterator so that $i.deposit$ is returned in x, and reorder the Iterator model components to give a more natural reading.

The representation of an Iterator as specified in Fig. 6 might look like the representation in Fig. 5. In addition, we have to store $i.deposit$ in the concrete representation; but this is accomplished simply by swapping the value in during Start_Iterator and swapping it back out during Finish_Iterator, so there are no substantive performance implications of this change.

Discussion: Here is a sample of client code for iteration using Design #3. (See the appendix for complete client examples.)

```
Start_Iterator(i, q, x)
maintaining i.past * i.future = #i.past
            * #i.future and
            i.present = x and
            i.original = #i.original and
            i.deposit = #i.deposit
while not Is_Empty (i) do
    Get_Next_Item(i, x)
    (* code to process x without changing
      i or x *)
end while
Finish_Iterator(i, q, x)
```

In this sample code, we include the loop invariant in a **maintaining** clause, which may be considered an extra syntactic slot in the while loop construct. The notation means that at the beginning of each iteration of the loop, the concatenation of $i.past$ and $i.future$ equals their concatenation just before the loop is first encountered; that $i.present$ equals x; and that $i.original$ and $i.deposit$ equal their respective values just before the loop is first encountered.

Clearly, this invariant is true at the start of the first iteration. It is easy to show that it is true for an arbitrary iteration if and only if the code to process x does not change i or x. With the addition of the past field to an Iterator's abstract state, it also is easy to show that all Items in the original Queue q, and only those Items, are processed by the loop.

The other changes in Design #3 support a general principle of the swapping paradigm: There are advantages in simplified reasoning about program behavior and in the performance of storage management activities if temporary variables in a program act as *catalyst* variables [9]. A catalyst variable is one that is necessary to carry out a computation, but experiences no (net) change in value from the beginning to the end of the computation, or is an initial value for its type at both points. In the expected use of Queue_Iterator_Template, we want to make sure the local variables i and x are catalysts. Notice that this is not the case for Design #2; i and x might start out with any values whatsoever before Start_Iterator, and their values after Finish_Iterator might be different.

In Design #3, we therefore require that Iterator i be an initial value for its type before the call to Start_Iterator. Finish_Iterator consumes i, leaving it again as an initial value for its type. Also in Design #3, we record the priming value of x that is passed to Start_Iterator and restore that value in Finish_Iterator; thus, the name $i.deposit$, reflecting the fact that we consider the priming value to be like a security deposit that should be returned to the client upon completion of the iteration. Now both i and x act as catalyst variables.

IV. Variations and Extensions

There are several interesting variations and extensions of this approach to iterators. We briefly discuss them here, and, in the process, propose a schema for a generic Iterator_Template concept (Fig. 7) that is flexible enough to accommodate most interesting uses for iterators. This concept schema constitutes our proposal for a common interface model for iterators.

```
concept Iterator_Template
    conceptual  context
        parametric  context
            type Item
            type Collection
    interface
        type  family Iterator is modeled by (
                past: string of math[Item]
                present: math[Item]
                future: string of math[Item]
                original:math[Collection]
                deposit: math[Item])
            exemplar i
            initialization
                ensures i.past = empty_string  and
                        is_initial (i.present)  and
                        i.future = empty_string  and
                        is_initial (i.original)  and
                        is_initial (i.deposit)
        operation Start_Iterator (
                alters i: Iterator
                consumes c: Collection
                consumes x: Item)
            requires    is_initial (i)
            ensures     i.past = empty_string  and
                        i.present = x  and
                        ρ (i.future, #c)  and
                        i.original = #c  and
                        i.deposit = #x
        operation Finish_Iterator (
                consumes i: Iterator
                produces c: Collection
                alters x: Item)
            requires    i.present = x
            ensures     c = #i.original  and
                        x = #i.deposit
        operation Get_Next_Item (
                alters i: Iterator
                alters x: Item)
            requires    i.present = x  and
                        i.future /= empty_string
            ensures     i.past = #i.past * <x>  and
                        i.present = x   and
                        <x> * i.future = #i.future  and
                        i.original = #i.original   and
                        i.deposit = #i.deposit
        operation Is_Empty (
                preserves i: Iterator): Boolean
            ensures     Is_Empty  iff  i.future = empty_string
end Iterator_Template
```

Fig. 7. Schema for a generic iterator design (with ρ free).

A. Early Exit from Iteration

A client program that exits from an iteration loop before the Iterator is empty poses no particular problem for Design #3. (See the Appendix for an example.) However, the rationale for implementing Queue_With_Iterator_Template as one module, and not layering the implementation of Queue_Iterator_Template on top of Queue_Template, is efficiency in this special case. If all the Queue_Template operations take constant time, then all the layered operations take constant time, except Finish_Iterator. In the case of an early exit from an iteration, Finish_Iterator takes time proportional to the number of Items remaining in the Iterator's future string. A direct implementation of Queue_With_Iterator_Template in which the Iterator operations have access to the underlying Queue representation (as in Fig. 5) achieves constant time performance for all operations.

B. Different Orders of Iteration and Iteration Over a Subset of All Items

It is easy to generalize the specification of Design #3 to define a schema for an Iterator type that presents the Items in a Queue to the client in a different order, and/or that iterates over just those Items that satisfy a particular condition. We define a binary (mathematical) relation:

$$\rho : \textbf{string of math}[\text{Item}] \times \textbf{string of math}[\text{Item}] \rightarrow \text{Boolean},$$

so that $\rho(s, t)$ holds whenever the order of appearance of the Items in string s is an acceptable or possible order of iteration for the desired Items in string t. We can now generalize the **ensures** clause of Start_Iterator as underlined.

```
operation Start_Iterator (
        alters i: Iterator
        consumes q: Queue
        consumes x: Item)
    requires  is_initial (i)
    ensures   i.past=empty_string and
              i.present = x and
              ρ(i.future,#q) and
              i.original= # q and
              i.deposit =# x
```

This operation specification, with ρ a free variable, should be interpreted as part of a schema for a concept, in the following sense. A specifier might use it to guide the design of different but related iterator concepts by binding ρ in any of three ways.

1) For each individual iterator concept, replace ρ by a particular relation that controls the order in which Items are to be returned by Get_Next_Item.
2) Make ρ a client-supplied parameter to the specification (like Item and Queue_Facility).
3) Make ρ an implementer-supplied parameter to the specification.

In cases (1) and (2), any realization must be further parameterized by program operations [4], [10] that satisfy certain properties involving ρ and that permit the implementer to write code that achieves the specified behavior. In case (3), the client knows only that ρ is some relation, possibly with additional mathematical properties dictated by the specifier. Here the implementer has the freedom to present the Items from the Iterator in any convenient (efficiently computed) order, and must supply a definition for ρ that characterizes the orders it might produce.

C. Other Collections

To specify Iterators for collections that are not modeled as mathematical strings, we can adapt the approach suggested above and parameterize the concept by a Collection type, as shown in Fig. 7. Again, we introduce a binary (mathematical) relation:

$$\rho\text{: } \textbf{string of math}[\text{Item}] \times \textbf{math}[\text{Collection}] \rightarrow \textbf{Boolean},$$

defined so that $\rho(s, c)$ holds whenever the order of the Items in string s is an acceptable or possible order of iteration for the desired Items in Collection c.

We need a relation here, not a function. Consider a Set ADT, where the mathematical model of a program Set is a mathematical set. Then a useful implementer-supplied relation ρ would have $\rho(s, c)$ hold exactly when every Item in set c occurs exactly once in string s. There is no natural order for iterating over the elements of a Set, but we probably want to specify that the iteration should see each element exactly once. If an implementer is free to choose any order that meets this criterion, there is substantial performance flexibility.

D. Modifying a Collection (But Not Its Items)

We now consider two more advanced kinds of iterators that involve modifying the collection during iteration. There are two sorts of changes: those that restructure the collection into an equivalent form without modifying any of its Items, and those that (instead or in addition) modify the values of the Items. An example of the first kind arises if we have a Tree ADT whose nodes are labeled by Items. We might not care about the shape of a Tree, as long as an in-order traversal produces the Items in the same order, e.g., if the Tree is used as a binary search tree. A side effect of iterating over such a Tree, then, might be that it is rebalanced.

How can we specify an iterator that has such an effect? We introduce another relation below:

$$\sigma\colon \textbf{math}[\texttt{Collection}] \times \textbf{string of math}[\texttt{Item}] \times \textbf{math}[\texttt{Collection}] \rightarrow \textbf{Boolean},$$

defined so that $\sigma(i, s, f)$ holds whenever the initial Collection i, when iterated over with the order of Items in string s, is equivalent to the final Collection f. We can then generalize the **ensures** clause of Finish_Iterator from the specification in Fig. 7, as underlined, below.

```
operation Finish_Iterator (
    consumes i: Iterator
    produces c: Collection
    alters x: Item)
 requires  i.present = x
 ensures   σ(#i.original, #i.past * #i.future, c)
           and x = #i.deposit
```

Now an implementation can return in c any Collection that is equivalent to the original Collection, offering the possibility of performance flexibility or even intentional restructuring. A degenerate case of this schema, where $\sigma(i, s, f)$ holds if and only if $\rho(s, i)$ holds and $i = f$, is the schema of Fig. 7.

E. Modifying the Items in a Collection

The intuitively obvious way to change every Item in a Collection is to iterate over the Collection and change each one as it is processed. Of course, this will not work directly with the proposed design, because getting the next Item requires the client to pass back exactly the same value that it received in the previous call to Get_Next_Item. This process works similarly for Finish_Iterator.

There are two ways to address this problem. One is to iterate over the Collection and construct the modified Collection as a new object. The appendix contains example code for copying a Queue in this way; there is no modification of each Item as it is added to the new Queue, but it is easy to see how this would be done if that were the objective. The difficulty with this as a general solution is that Items cannot be modified in place. New ones must be constructed, with the associated efficiency penalty (which is possibly significant if the Items are large) that we initially argued we should like to avoid if possible.

Another approach, then, is to further generalize the Iterator_Template design to support specifying the way in which each Item is to be modified. Again, we introduce a relation that characterizes mathematically how the modified Item values must be related to the old ones:

$$\nu: \textbf{math}\,[\texttt{Item}] \times \textbf{math}\,[\texttt{Item}] \rightarrow \textbf{Boolean}.$$

The specifier or client should define $\nu(a, b)$ to hold if and only if b is an allowable new value corresponding to the old value a. (This relation could be generalized even further to have a third argument, a string of Items, so that new Item values could depend on the values of all previously processed Items as well.)

Now we generalize the preconditions of Get_Next_Item and Finish_Iterator, replacing `i.present = x` by ν `(i.present, x)`. We also have to do two other things. The first is to add another component to the Iterator model—a string of Items perhaps called updates—and to change the postcondition of Get_Next_Item, so that this string records the (modified) Items returned to Get_Next_Item. The second is to change the relation σ from the previous subsection, so that it depends additionally on another string of Items that includes the updates, and to change the postcondition of Finish_Iterator accordingly.

Note that modifying a collection while iterating over it, though specifiable and sometimes useful, is fraught with danger. Consider iterating over a Set of Integers, squaring each one. The abstract set model of the program Set object might have fewer elements following the iteration; e.g., -2 and 2 both yield 4. Similarly, consider iterating over a Tree of Integers, squaring each one, but trying to maintain the binary search tree property. These examples illustrate that for a correct implementation, it is insufficient just to traverse the data structure representing the Set or Tree and to perform a squaring operation on each element. This is the kind of problem, both in client understanding of iteration and in implementation efficiency, that leads us to warn against modifying Items during iteration in general, even though it causes no insurmountable technical difficulties with our specification and design approach.

V. Conclusion

Previously published iterator designs are unsatisfactory along several dimensions. The iterator design developed incrementally for Queues in Section III, and generalized to a schema for arbitrary Collections in Section IV-C, addresses the deficiencies of prior approaches in the following specific ways.

- It is designed to support efficient implementations; neither the implementer nor the client needs to copy the data structure representing the Collection, or any of the individual Items in it.

- Its abstract behavior (including the noninterference property) is formally specified.
- Its implementations and clients can be verified independently, i.e., modularly, in the sense of [8].
- It can be specified as a schema for an independent generic concept that defines an iterator abstraction for arbitrary Collections, so all iterator abstractions in a system share a common interface model.

Because of these advantages, the iterator design in Fig. 7 should be considered as a baseline proposal for a common interface model for iterator abstractions. This baseline supports sequential access to the individual Items of a Collection in various orders, but without allowing a Collection or its Items to be modified during iteration, and is robust enough to handle any container structure where such iterations are meaningful.

A final note on language issues: Our design shows, in principle, how iterators can be abstracted and encapsulated to support modular programming and modular reasoning about program behavior. But can the design be used in real programming languages? There is no technical difficulty with Ada, because a generic package may export more than one type, as an implementation of a Collection_With_Iterator_Template must. (See the *RESOLVE/Ada Discipline* [10]. The particularly interested reader also should consult [4] for detailed examples of similar iterator designs coded in Ada.)

For C++, a mismatch with the RESOLVE language model leads to minor trouble. There is a temptation to use inheritance to combine interfaces, i.e., to make Queue_With_Iterator_Template a class derived from the Queue_Template class. However, such a C++ class effectively defines just one type, not two. This leads inevitably to nontrivial differences between the abstract specification given here and even the parameter profiles of the C++ class methods. So, another solution is preferred: Make Queue_Template and Queue_Iterator_Template separate but friend classes in order to get the required efficiency of implementation of the combined interface.

Appendix
Client Examples for Design #3

Here is a sample client for Design #3, an operation to copy a queue using an iterator.

```
operation Copy (
        preserves q1: Queue
        produces q2: Queue)
    ensures q2 = q1
begin
    variable
        cleared: Queue
        i: Iterator
        x1, x2: Item

    q2 :=: cleared
    Start_Iterator (i, q1, x1)
    maintaining i.past * i.future =
                #i.past * # i.future and
                i.present = x1 and
                i.original = #i.original
                  and
                i.deposit = #i.deposit
                  and
                q2 = i.past
    while not Is_Empty (i) do
            Get_Next_Item (i, x1)
            Copy_Item (x1, x2)
            Enqueue (q2, x2)
    end while
    Finish_Iterator (i, q1, x1)
end Copy
```

This example illustrates simultaneous iteration over two collections, and a possible early exit from an iteration loop: an operation to determine whether two Queues are equal.

```
operation Are_Equal (
        preserves q1: Queue
        preserves q2: Queue): Boolean
    ensures Are_Equal iff q1 = q2

begin
    variable
        i1, i2: Iterator
        x1, x2: Item
        equal: Boolean

    equal := true
    Start_Iterator (i1, q1, x1)
    Start_Iterator (i2, q2, x2)
    maintaining i1.past * i1.future =
                  #i1.past * #i1.future
                  and
                i1.present = x1 and
                i1.original =
                #i1.original and
                i1.deposit= #i1.deposit)
                  and
                i2.past * i2.future =
                  #i2.past * #i2.future
                  and
                i2.present = x2 and
                i2.original =
                  #i2.original and
                i2.deposit = #i2.deposit
                  and
                equal = (i1.past =
                  i2.past)
    while equal and not Is_Empty (i1) and
      not Is_Empty (i2) do
            Get_Next_Item (i1, x1)
            Get_Next_Item (i2, x2)
            equal := Are_Equal_Items (x1,
              x2)
    end while
    if equal and (not Is_Empty (i1) or not
        Is_Empty (i2)) then
            equal := false
      end if
      Finish_Iterator (i1, q1, x1)
      Finish_Iterator (i2, q2, x2)
      return equal
  end Are_Equal
```

ACKNOWLEDGMENT

We are indebted to W. Heym, J. Hollingsworth, B. Ogden, M. Sitaraman, S. Zweben, and the anonymous referees for many helpful comments.

REFERENCES

[1] J. M. Bishop, "The effect of data abstraction on loop programming techniques," *IEEE Trans. Software Eng.*, vol. 16, pp. 389–402, Apr. 1990.

[2] G. Booch, *Software Components with Ada.* Redwood City, CA: Benjamin-Cummings, 1987.

[3] R. D. Cameron, "Efficient high-level iteration with accumulators," *ACM TOPLAS*, vol. 11, pp. 194–211, Apr. 1989.

[4] S. H. Edwards, "An approach for constructing reusable software components in Ada," Tech. Rep. P-2378, Inst. for Defense Analyses, Alexandria, VA, USA, 1990.

[5] ——, "Common interface models for components are necessary to support composability," *Proc. 4th Ann. Workshop on Software Reuse*, SPC, Herndon, VA, USA, 1991.

[6] ——, "Common interface models for reusable software," *Int. J. Software Eng. Knowledge Eng.*, vol. 3, pp. 193–206, June 1993.

[7] ——, "A formal model of software subsystems," Ph.D. dissertation, Dept. of Comput. and Inform. Sci., Ohio State Univ., Columbus, OH, USA, in preparation.

[8] G. W. Ernst, R. J. Hookway, and W. F. Ogden, "Modular verification of data abstractions with shared representations," *IEEE Trans. Software Eng.*, vol. 20, pp. 288–307, Apr. 1994.

[9] D. E. Harms and B. W. Weide, "Copying and swapping: Influences on the design of reusable software components," *IEEE Trans. Software Eng.* vol. 17, pp. 424–435, May 1991.

[10] J. E. Hollingsworth, "Software component design-for reuse: A language independent discipline applied to Ada," Ph.D. dissertation, Dept. of Comput. and Inform. Sci., Ohio State Univ., Columbus, OH, USA, 1992.

[11] W. R. LaLonde, "Designing families of data types using exemplars," *ACM Trans. Programming Languages Syst.*, vol. 11, pp. 212–248, 1989.

[12] D. A. Lamb, "Specification of iterators," *IEEE Trans. Software Eng.*, vol. 16, pp. 1352–1360, Dec. 1990.

[13] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction mechanisms in CLU," *CACM*, vol. 20, no. 8, pp. 564–576, Aug. 1977.

[14] S. Muralidharan and B. W. Weide, "Should data abstraction be violated to enhance software reuse?" *Proc. 8th Ann. Natl. Conf. Ada Technol.*, 1990, pp. 515–524.

[15] T. W. Pearce and D. A. Lamb, "The property vector specification of a multiset iterator," *Proc. 14th Int. ACM/IEEE Conf. Software Eng.*, 1992.

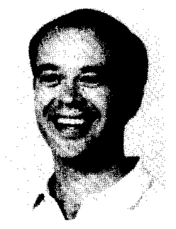
[16] M. Shaw, W. A. Wulf, and R. L. London, "Abstraction and verification in Alphard: defining and specifying iteration and generators," *CACM*, vol. 20, no. 8, pp. 553–564, Aug. 1977.

[17] M. Sitaraman, L. R. Welch, and D. E. Harms, "On specification of reusable software components," *Int. J. Software Eng. Knowledge Eng.*, vol. 3, pp. 207–229, June 1993.

[18] W. Tracz, "Parameterization: A case study," *Ada Lett.*, vol. 9, pp. 92–102, May/June 1989.

[19] B. W. Weide, W. F. Ogden, and S. H. Zweben, "Reusable software components," in M. C. Yovits, Ed., *Advances in Computers*, vol. 33. New York: Academic, 1991, pp. 1–65.

[20] J. M. Wing, "A specifier's introduction to formal methods," *Comput.*, vol. 23, pp. 8–24, Sept. 1990.



B. W. Weide (S'73–M'78) received the B.S.E.E. degree from the University of Toledo, OH, USA, and the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, USA.

He is an Associate Professor of Computer and Information Science at Ohio State University, Columbus, OH, USA, and Codirector of the Reusable Software Research Group with Bill Ogden and Stu Zweben. His research interests include all aspects of software component engineering, especially in applying RSRG work to Ada and C++ practice.

Dr. Weide is a member of the IEEE, ACM, and CPSR.

S. H. Edwards received the B.S.E.E. degree from the California Institute of Technology and the M.S. degree in computer and information science from Ohio State University, Columbus, OH, USA.

He is a Ph.D. degree candidate in computer and information science at Ohio State University. Prior to attending Ohio State, he was a Member of Research Staff at the Institute for Defense Analyses. His research interests are in software engineering and reuse, formal models of software structure, programming languages, and information retrieval technology.

Mr. Edwards is a member of the IEEE Computer Society and ACM.

D. E. Harms (S'87–M'88) received the B.S. from Muskingum College, New Concord, OH, USA, and the M.S. and Ph.D. degrees from Ohio State University, Columbus, OH, USA.

He is an Associate Professor of computer science at Muskingum College. His research interests are in software engineering (especially reuse, specification, and verification) and programming language design.

Dr. Harms is a member of the IEEE and ACM.

D. A. Lamb (S'75–M'77–SM'87) received the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, USA, in 1983.

He is an Associate Professor of Computing and Information Science at Queen's University, Kingston, ON, Canada. His research interests include software design methods, configuration management, and formal methods in software engineering.

Dr. Lamb is a member of ACM and Sigma Xi.