SAVCBS 2006 Challenge: Specification of Iterators

Bruce W. Weide
Department of Computer Science and Engineering
The Ohio State University
+1-614-292-1517
weide.1@osu.edu

ABSTRACT
A method for formal specification of iterators, which can be used to verify both clients and implementations, is illustrated with a *Set* abstraction as the underlying collection.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/program verification.

General Terms
Design, Verification.

Keywords
Formal specification, iterators, Resolve, verification.

1. INTRODUCTION
This short paper is a response to the SAVCBS 2006 Challenge Problem: "We invite participants to illustrate their specification and verification techniques on the problem of specifying the behavior of iterators and clients that use them". Our solution illustrates, and slightly improves on (i.e., simplifies) the iterator design and specification techniques we previously published in [8], using a *Set* abstraction with an active iterator that does not permit interleaved client modification of the elements of a *Set*. The conclusion of [8] is:

> Previously published iterator designs are unsatisfactory along several dimensions. The iterator design developed incrementally [in this paper] 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 non-interference property) is formally specified.
> - Its implementations and clients can be verified independently, i.e., modularly in the sense of [3].
> - 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.

Fifth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2006). November 10-11, 2006, Portland, Oregon, USA.

"Non-interference" means [8] that 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." Extensions and variants discussed in [8] also address issues involved when iteration over a collection modifies the collection or the items in it, when iteration might encounter items in different orders, and when iteration terminates early. None of these latter issues is explicitly discussed here.

The answers to the specific questions posed in the Challenge Problem are as follows:

- The solution is intended for use with a sequential programming language, though concurrency-hardening does not seem to pose any special problems.
- The level of annotation required (both in the contract specification and in a client program) is full behavioral specification—but no more than what is necessary and sufficient to modularly verify client correctness. In principle, it might be possible to specify or prove weaker properties with less annotation, but we see no reason to do so; this solution seems fully manageable in terms of specification and verification complexity.
- The solution is based on using a language, such as Resolve [2] or the disciplined use of C++ that we call Resolve/C++ [6], that has value semantics, with no visible references and hence no visible aliasing. We emphasize that this does not imply inefficiency compared to languages that make references manifest to the programmer. This is one of the main points of [8]—but one that we do not elaborate here except to claim the result that our design permits optimally efficient iterator implementations in the big-O sense, so introducing the reasoning complications of reference semantics would not result in efficiency improvements.
- Fully automated verification of client programs using our iterator design and specification approach is certainly possible in principle. We know how to generate mechanically the verification conditions for Resolve programs. However, there is no evidence yet to suggest that a system like Hoare's "verifying compiler", that would produce fully automatic proofs of these verification conditions, is just over the horizon. The verification conditions that arise from using our iterators are not particular difficult for humans to discharge. They do seem generally near or beyond what existing theorem provers can handle without human advice.

We regrettably have no fundamentally new observations about specifying iterators since the 1994 paper [8]. We hope the attendees at the workshop provide some additional food for thought.

2. EXAMPLE: A SET COMPONENT

Understanding our iterator design and specification requires understanding the specification of the collection over which iteration is to be done. The iterator design technique we proposed in [8] is a schema that can be used with arbitrary collections and is illustrated there with a *Queue* abstraction. Here we use for variety a *Set* abstraction: a parameterized component in which the type *Item* (of a *Set*'s elements) is a template parameter. This is its specification:

```
contract Set_Template (type Item)
  type family Set is modeled by
      finite set of Item
    exemplar s
    initialization ensures
      s = { }
  operation Add (s: Set, x: Item)
    requires
      x is not in s
    ensures
      s = #s union {#x}
  operation Remove (s: Set, x: Item,
       x_copy: Item)
    requires
      x is in s
    ensures
      x = x_copy = #x  and
      s = #s - {x_copy}
  operation Remove_Any (s: Set, x: Item)
    requires
      s /= { }
    ensures
      s = #s - {x}
  operation Is_Member (s: Set, x: Item):
       Boolean
    ensures
      Is_Member = (x is in s)
  operation Size (s: Set): Integer
    ensures
      Size = |s|
end Set_Template
```

The type specification says that a *Set* variable should be considered to have a value that is a finite mathematical set of the parameter type *Item*, and that such a variable's initial value (i.e., upon declaration) is an empty set. The operation *Add* can be used to add an element to a *Set*; the operation *Remove* to remove a particular element whose value is *x*, the removed element being returned in *x_copy*; the operation *Remove_Any* to remove and return an arbitrary implementation-determined element, which is needed for functional completeness of this component [7] in the absence of an accompanying iterator; the function operation *Is_Member* to test set membership; and the function operation *Size* to determine the number of elements. In operation specifications, the prefix "#" on a variable name in a postcondition denotes the parameter's incoming value. Function operations may not change the values of their arguments, so this fact is not specified explicitly.

Two important points must be kept in mind. First, there are no hidden references here. The simplicity of the specification is the result neither of hoping/assuming that there are no aliases (i.e., aliases really aren't possible), nor of syntactic sugar that makes *s*, *x*, *#s*, etc., simply appear to act like mathematical variables rather than the names of objects (i.e., they really do act like mathematical variables). This might surprise readers who are used to similar-looking specifications that deal with references. Second, the only other operations that are available for a *Set* type are those available for any type in Resolve: *Clear*, which resets a variable to an initial value for its type, and a swap operator ":=:" that exchanges the values of two variables of the same type [8]. Assignment ":=" is available only when the right-hand side is a call to a function operation. Readers who are unfamiliar with this style of programming under design-by-contract—value types only, built-in swapping but not variable-to-variable assignment, fully parameterized components, etc.—also might be surprised to learn that it is possible and practical to develop "real" software this way with disciplined use of C++. In fact, experience with a commercial Windows application of over 100,000 SLOC developed in this style shows that real software is not only feasible but also of notably higher quality than software built using traditional methods [4]. In other words, this proposal for how to specify iterators is not based on an unrealistic closed-world assumption; it also is not based on business-as-usual in C++ or Java.

3. CLIENT AND COMPONENT DESIGN

In our design, an iterator's abstract model value includes a string of *Item*s called *past*, which is essentially the *Item*s iterated over so far (in the order they have been processed, with the value of the *Item* currently "out" of the collection at the end of this string), a string of *Item*s called *future*, i.e., those to be iterated over in the future (in the order to be processed, unless the iteration terminates early), and a set called *original*, which is the original value of the *Set* over which iteration is being done. For the complete rationale behind this style of design, see [8]. The formal specification is on the next page (Section 5). Here is a fragment of a typical client program that iterates completely over the *Set s*:

```
Start_Iterator (i, s, x)
loop
  maintaining
    i.past * i.future =
      #i.past * #i.future  and
    <x> is suffix of i.past  and
    i.original = #i.original
  decreasing
    |i.future|
while Length_Of_Future (i) > 0 do
  Get_Next_Item (i, x)
  /* process x, with no net change to it */
end loop
Finish_Iterator (i, s, x)
```

Bracketing calls to *Start_Iterator* and *Finish_Iterator* move the elements of the original *Set s* into the *Set_Iterator i*

and back again. This prevents interference between iteration over *i* and interleaved modifications to *s*: client code that does this might be useless because the changes to *s* are lost when *Finish_Iterator* executes, but that client is not necessarily incorrect. Users of C++ or Java or other similar iterators might find this behavior unsettling. However, possible interference between interleaved modifications to a collection and iteration over it leads to informal and difficult-to-specify warnings in component libraries, such as this one for the *remove* method in the java.util package, interface *Iterator<E>* [5]:

> The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

We did not feel obligated to keep such problematic behavior regardless of its familiarity (in 2006 even more so than in 1994), opting instead for simpler, easily explicable behavior that can be specified without introducing either new specification constructs or extra-specificational warnings.

Start_Iterator records the value of *x* in the string *i.past* at the start of the loop—important to meet the precondition of the first call of *Get_Next_Item*. Parsimony, as well as a Resolve design rule urging "conservation of data", suggest that eventually *x* should have this value again after completion of *Finish_Iterator*. The loop invariant is that the concatenation of *i.past* and *i.future* does not change, that *i.original* does not change, and that the last entry in *i.past* equals *x*. Of course, there is more to the loop invariant to prove the correctness of what the client program is doing while iterating over the elements of *s*, but this is the part required to show that the iterator *i* is being used properly.

Given the stylized nature of the client code, it is easy to imagine special iteration syntax for collections, such as that now available in Java, but with a semantics that matches this common interface model [1] for iterators rather than Java's *Iterable<T>* and *Iterator<E>* interfaces.

4. REFERENCES

[1] Edwards, S.H., "Common Interface Models for Reusable Software", *Intl. J. of Softw. Eng. and Knowledge Eng. 3*, 2 (June 1993), 193-206.

[2] Edwards, S.H., Heym, W.D., Long, T.J., Sitaraman, M., and Weide, B.W., "Specifying Components in RESOLVE," *Software Eng. Notes 19*, 4 (October 1994), 29-39.

[3] Ernst, G.W., Hookway, R.J., and Ogden, W.F., "Modular Verification of Data Abstractions with Shared Realizations", *IEEE TSE 20*, 4 (Apr 1994), 288-207.

[4] Hollingsworth, J.E., Blankenship, L., and Weide, B.W., "Experience Report: Using RESOLVE/C++ for Commercial Software", *Proc. ACM SIGSOFT 8th Intl. Symp. on the Foundations of Softw. Eng.,* ACM Press, 2000, 11-19.

[5] java.util Package, *Interface Iterator <E>, remove Method Detail*, http://java.sun.com/j2se/1.5.0/docs/api/java/util/-Iterator.html, viewed 6 Oct. 2006.

[6] Resolve/C++, http://www.cse.ohio-state.edu/sce/now, viewed 6 Oct. 2006.

[7] Weide, B.W., Ogden, W.F., and Zweben, S.H., "Reusable Software Components", in *Advances in Computers*, vol. 33, M.C.Yovits, ed., Academic Press, 1991, 1-65.

[8] Weide, B.W., Edwards, S.H., Harms, D.E., and Lamb, D.A., "Design and Specification of Iterators Using the Swapping Paradigm," *IEEE TSE 20*, 8 (August 1994), 631-643.

5. APPENDIX: THE SPECIFICATION

```
contract Set_With_Iterator_Template
    enhances Set_Template
  type family Set_Iterator is modeled by (
       past: string of Item,
       future: string of Item,
       original: finite set of Item
      )
    exemplar i
    initialization  ensures
      i = (< >, < >, { })
  operation Start_Iterator (i: Set_Iterator,
       s: Set, x: Item)
    ensures
      there exists f: string of Item
        (elements (f) = #s  and
         |f| = |#s|  and
         i = (<x>, f, #s))  and
      s = { }  and
      x = #x
  operation Finish_Iterator (
       i: Set_Iterator, s: Set, x: Item)
    requires
      <x> is suffix of i.past
    ensures
      is_initial (i)  and
      s = #i.original  and
      <x> is prefix of #i.past
  operation Get_Next_Item (i: Set_Iterator,
       x: Item)
    requires
      i.future /= < >  and
      <x> is suffix of i.past
    ensures
      there exists f: string of Item
        (#i.future = <x> * f  and
         i = (#i.past * <x>, f, #i.original))
  operation Length_Of_Future (
       i: Set_Iterator): Integer
    ensures
      Length_Of_Future = |i.future|
end Set_With_Iterator_Template
```