# Specifying Java Iterators with JML and Esc/Java2

David R. Cok
Eastman Kodak Company
1999 Lake Avenue Rochester, NY 14650, USA
david.cok@kodak.com

## Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Program Verification

## General Terms

Design, verification

## Keywords

JML, ESC/Java2, static analysis, specification, verification

## 1. INTRODUCTION

The 2006 SAVCBS Workshop[1] has posed a Challenge Problem on the topic of specifying iterators. This note provides a specification in the Java Modeling Language (JML) [1, 2] for the Java interfaces *Iterator* and *Iterable* that captures the interactions between these two interfaces. An example program that uses these interfaces is checked using Esc/Java2 [3, 4, 5], demonstrating by example that the Esc/Java2 tool checks that the interfaces are used only as required by the specifications. The concluding section contains some observations on the limitations of JML for this specification task.

## 2. THE PROBLEM

The Challenge Problem[2] asks for a specification of the *Iterator* interface as provided in the Java programming language or its equivalent in another language. An Iterator provides an abstract mechanism for sequentially retrieving the elements of an object for which such an operation is appropriate, that is, of an *Iterable* object. There are two aspects of the behavior of an iterator.

The first is the mechanism for keeping track of which objects of the iterable collection have already been returned by the iterator and which are yet to be returned. This mechanism is dependent on the particular kind of iterable object (e.g., set, array, list, infinite sequence) and its implementation. In fact there is actually quite little one can specify about this aspect of an iterator's behavior within the *Iterator* interface itself. Space limitations preclude discussing the specification of that mechanism here.

The more interesting aspect of an iterator's behavior is the interaction among multiple iterators and with the iterable object, particularly with respect to modifications of the iterable object. In particular, the solution presented here provides specifications for three conditions: (a) an iterator may remove the object of the iterable at the current position of the iterator, but it may not remove it more than once; (b) if an iterable object is modified by its own methods, then all subsequent behavior of iterators referring to that iterable is undefined; (c) if an iterable object is modified by an iterator, then all subsequent behavior of any other iterator referring to that iterable is undefined.

Here we consider only sequential programs and provide a solution for Java 1.5 using JML. The reader is presumed to be familiar with Java and its iterator classes as well as with JML. In actuality, JML is implemented only for Java 1.4. However, the only use of Java 1.5 features here is the parameterization of the interfaces by the element type *E*, and that does not affect the discussion below. On the other hand, Java 1.4 does not have the equivalent of the *Iterable* interface, a point that is discussed further below.

## 3. THE JML SPECIFICATION

The proposed specifications of these two interfaces are shown in Figs. 1 and 2. A partial specification of the *Collection* interface is also shown.

The solution has the following elements:

- Because the conditions above require certain behavior *subsequent to* other behavior, a concept of time (or, more precisely, of an ordered sequence of events) is maintained in the specification by nondecreasing integer values.
- An *Iterator* maintains a reference to the *Iterable* whose contents it returns, contained in the model field `iterable`. This field is initialized at construction time (in the method `Iterable.iterator()`) and does not change thereafter, as indicated by the constraint clause.
- An *Iterator* uses the model field `iteratorTime` to keep track of when it was created or last used to modifiy the iterable. An initial value is specified by the

[1] http://www.cs.iastate.edu/~leavens/SAVCBS/2006/index.shtml.
[2] http://www.cs.iastate.edu/~leavens/SAVCBS/2006/challenge.shtml.


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

```
package java.util;
public interface Iterator<E> {
  //@ public instance model Iterable iterable;
  //@ public instance model int iteratorTime;
  //@ public instance model boolean removeOK;

  //@ initially !removeOK;
  //@ public invariant iterable != null;
  //@ public constraint iterable == \old(iterable);

  // This returns false if the parent Iterable has
  // been modified by means other than this Iterator.
  /*@ public normal_behavior
    @  ensures \result ==
    @      (iteratorTime > iterable.lastModifiedTime);
    @ public pure model boolean isValid() {
    @  return iteratorTime > iterable.lastModifiedTime;
    @ }
    @*/

  //@ public normal_behavior
  //@  requires isValid();
  /@ pure @*/ public boolean hasNext();

  //@ public normal_behavior
  //@  requires isValid() && hasNext();
  //@  assignable removeOK;
  //@  ensures removeOK;
  //@ also public exceptional_behavior
  //@  requires isValid() && !hasNext();
  //@  signals_only NoSuchElementException;
  public E next();

  /*@ public behavior
    @  requires isValid() && removeOK;
    @  assignable removeOK, iteratorTime;
    @  assignable iterable.maxIteratorTime;
    @  assignable iterable.lastModifiedTime;
    @  ensures !removeOK;
    @  ensures iterable.lastModifiedTime >
    @              \old(iterable.maxIteratorTime);
    @  ensures isValid();
    @  ensures iteratorTime <= iterable.maxIteratorTime;
    @ also public exceptional_behavior
    @  requires isValid() && !removeOK;
    @  signals_only IllegalStateException;
    @*/
  public void remove();
}
```

**Figure 1: The specification of the Iterator interface.**

method `Iterable.iterator()` and it is modified only by `Iterator.remove()`.

- The model field `Iterator.removeOK` indicates whether it is permissible to call the method `Iterator.remove()`. The field is initially false and is also set false upon any call of `remove`; it is set true on a call of `next`. Thus informal requirement (a) above is satisfied.

```
package java.lang;
public interface Iterable<E> {
  //@ public instance model int lastModifiedTime;
  //@ public instance model int maxIteratorTime;
  //@ initially maxIteratorTime == -1;
  //@ initially lastModifiedTime == 0;
  /*@ constraint lastModifiedTime >=
                          \old(lastModifiedTime); @*/

  //@ public normal_behavior
  //@   assignable maxIteratorTime;
  //@   ensures \result != null;
  //@   ensures \fresh(\result);
  //@   ensures \result.iterable == this;
  //@   ensures \result.isValid();
  //@   ensures maxIteratorTime >= \result.iteratorTime;
  public Iterator<E> iterator();
}

package java.util;
public interface Collection<E> extends Iterable<E> {
  // Something like the following specification
  // case must be present for any method that
  // modifies the Iterable object.
  //@ public normal_behavior
  //@   assignable lastModifiedTime;
  //@   ensures lastModifiedTime > maxIteratorTime;
  public void clear();
}
```

**Figure 2: The specification of the Iterable interface and a partial specification of Collection.**

- Requirements (b) and (c) above need the distinction between an Iterator's behavior being defined and not defined. This distinction is provided by the pure model method `Iterator.isValid()`. If the method returns true, the behavior is defined. The method is implemented to return true if the iterator's `iteratorTime` is larger than the corresponding iterable's `lastModifiedTime`.

- An *Iterable* maintains the "time" of its last modification in the field `lastModifiedTime`. If the *Iterable* is modified, as shown by the method `Collection.clear`, the value of `lastModifiedTime` is increased to be larger than the `iteratorTime` of any of its associated *Iterator*s. For convenience, `Iterable.maxIteratorTime` holds a value at least as large as any associated *Iterator*'s `iteratorTime`. This satisfies requirement (b) above. Note that any method in any subtype of `Iterable` that modifies the collection of elements within the Iterable (e.g., add, remove, clear) must require a specification case like that shown for `Collection.clear`.

- Requirement (c) is satisfied as follows. The specification of `Iterator.remove` requires that when called on an object *iter* (and for normal termination), the corresponding iterable's `lastModifiedTime` is increased to make all other iterators invalid, and the `iteratorTime` of *iter* itself also is increased so that *iter* is still valid.

# 4. STATICALLY CHECKING PROGRAMS USING ESC/JAVA2

The *Iterator* and *Iterable* interfaces do not have implementations that can be checked against specifications. However, we can check programs that use those interfaces. To do so with JML and Esc/Java2, however, we must recast the above solution in Java 1.4. For this exercise we fold the specifications from *Iterable* into Java 1.4's *Collection* interface. Then we attempt to check a number of combinations of uses of these methods, as shown in Figs. 3 and 4. Esc/Java2[3] successfully finds the incorrect uses of these methods and has no false reports on legal sequences of method calls. The problems in generating and checking the specifications were all in specification errors (not in Esc/Java2). For example, in method `m6`, if Line A is omitted, allowing aliasing between the two arguments (a common error), then Line B cannot be established: iterator `ii` will not be valid if `c==cc`.

# 5. OBSERVATIONS

The combination of JML and Esc/Java2 successfully specifies the Iterator example and checks uses of the interfaces in test programs. However, this exercise prompts a number of observations about the current state of JML.

## 5.1 Java 1.4 vs. Java 1.5

This style of solution will not work well in Java 1.4 because there is no abstract *Iterable* object. For the static checking above, we utilized the *Collection* interface as the generic iterable. However, not all iterators extend the *Collection* interface. Thus in Java 1.4 an *Iterator* can only refer to its associated object as a generic *Object*, and there is no place to put the declarations of the model fields defined above. An alternative, but messy, design is to declare a new associated `IterableData` class containing the model fields declared above in *Iterable* and used as *Iterable* is above; then we associate an `IterableData` object with each iterable `Object` by maintaining a Map from objects that would be Iterables to associated instances of `IterableData`.

## 5.2 Ghost field vs. Model field vs. Model method

In the specification above, various pieces of specification information are held in model fields. These might also be declared as ghost fields or model methods. Each of these choices has its disadvantages.

- Ghost fields. Iterators and Iterables are interfaces, not classes. Furthermore, they are defined in the Java library and not in user-written code. Ghost fields must be modified by JML `set` statements within the implementation of a method. In this situation, for these interfaces there is no place to put those `set` statements. This is not a problem for static checking, but runtime checking (such as with the jmlrac[2] tool) would fail to work correctly if ghost fields were used.

- Model fields. The intended use of a model field is as a means to hold an abstract representation of the state of an object; in a concrete class each model field would be provided a representation. In this case, a field such as `removeOK` does abstract part of the state of the Iterator, but that abstraction is not necessarily a representation of any concrete fields of an implementation. A typical way to provide such a concrete representation is by means of some ghost fields that essentially duplicate the model fields. The model fields work well for static checking without ghost fields and without representations. However, runtime checking would require the model fields to have some concrete representation.

- Model methods. Model methods are an alternate way of providing the functionality of a model field.[4] For example, instead of the field `removeOK`, we could have a pure, argument-less model method `removeOK()` without any implementation given. The specification of its

[3]The experiments were performed using the version in CVS HEAD as of 1 September 2006, but only using the specifications given here, not the library of system specifications provided by Esc/Java2.

[4]Model fields also have implications for data groups, which model methods do not have.

```
import java.util.Collection;
import java.util.Iterator;
public class Test {
  public void m1(/*@ non_null @*/Collection c) {
    Iterator i = c.iterator();
    i.remove();  // should FAIL
  }

  //@ signals (java.util.NoSuchElementException);
  //@ signals_only RuntimeException;
  public void m2(/*@ non_null @*/Collection c) {
    Iterator i = c.iterator();
    //@ assume i.hasNext();
    i.next();
    i.remove();  // OK
  }

  public void m3(/*@ non_null @*/Collection c) {
    Iterator i = c.iterator();
    //@ assume i.hasNext();
    i.next();
    i.remove();
    i.remove();  // should FAIL
  }

  public void m4a(/*@ non_null @*/Collection c) {
    Iterator i = c.iterator();
    //@ assert i.iteratorTime > c.lastModifiedTime;
    //@ assert i.iterable == c;
    //@ assert i.isValid();
  }

  public void m4(/*@ non_null @*/Collection c) {
    Iterator i = c.iterator();
    //@ assert i.isValid();
    c.clear();
    //@ assert !i.isValid();
  }
}
```

**Figure 3: A set of test methods (in Java 1.4).**

```
import java.util.Collection;
import java.util.Iterator;
public class Test2 {
  public void m5(/*@ non_null @*/Collection c) {
    Iterator i = c.iterator();
    Iterator ii = c.iterator();
    //@ assert i.isValid();
    //@ assert ii.isValid();
    //@ assume i.hasNext();
    i.next();
    i.remove();
    //@ assert i.isValid();
    //@ assert !ii.isValid();
  }

  //@ requires c != cc;     // Line A
  public void m6(/*@ non_null @*/Collection c,
                 /*@ non_null @*/Collection cc) {
    Iterator i = c.iterator();
    Iterator ii = cc.iterator();
    //@ assert i.isValid();
    //@ assert ii.isValid();
    //@ assume i.hasNext();
    i.next();
    i.remove();
    //@ assert i.isValid();
    //@ assert ii.isValid();  // Line B
  }

  public void m7(/*@ non_null @*/Collection c) {
    Iterator i = c.iterator();
    //@ assume i.hasNext();
    c.clear();
    i.hasNext();//FAILS - precondition isValid()
  }             // is not satisfied
}
```

**Figure 4: Additional test methods (in Java 1.4).**

result and its use on other specifications would mimic the specification and use of the model field. Static checking with such model methods is just as easy (and as hard) as when using model fields. Runtime checking has the same problems as with model fields: we need an implementation in terms of concrete or ghost fields.

One enhancement of JML that would help the above issues with runtime checking would be to provide syntax in which updates to ghost fields could be specified and compiled by a runtime checker even for methods for which the runtime checker did not compile the Java implementation of the method itself.

## 5.3 Specifying mutating methods

As stated earlier, the specification described here requires that all methods (of any subtype) that modify an *Iterable* object must specify that the values of `lastModifiedTime` and `maxIteratorTime` are appropriately changed. This requirement is easily forgotten. Any method that calls `remove()` will encounter those requirements in that method's specification, but other methods, such as `add`, will not. Aside from the specifications of overridden methods, there is no way within JML to require that all methods with certain properties have certain specifications without individually annotating the methods to indicate the desired property.

## 5.4 Specifying sequences of calls

The main limitation of JML in this context is that it provides no means to write specifications about sequences of method calls. The specification above essentially encodes two state machines: a simple one using `removeOK` and a more complicated one involving the other model fields. These machines are used to specify implicitly the behavior of sequences of method calls. However, there is no way in JML to write a specification requirement about this behavior that can be checked by some reasoning engine; in Section 3 we were only able to argue the correctness of the specifications informally. The best we can do in current JML is to write example programs and then check using a static checker that those examples are properly handled; that process, like runtime testing, does not ensure that all possible examples will behave correctly. Another common restriction is when a class has an initialization method that must be called before any other method of the class is called.

To express these conditions, JML would need to have syntax that could encode, for example, the following requirements: that two calls of `Iterator.remove` with no intervening call of `Iterator.next` must result in particular behavior; that a call of a class method not preceded by a call of the class's init method results in an exception being thrown; that a call of a particular method will render calls of another set of methods undefined. These all would require syntax enabling the expression of combinations of parameterized sequences of method calls, with options such as are found in regular expressions. In addition, we would need translation to verification conditions in an appropriate logic and suitable for a logical prover.

# 6. REFERENCES


[1] Many references to papers on JML can be found on the JML project website, http://www.cs.iastate.edu/~leavens/JML/papers.shtml.

[2] L. Burdy, et al. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, June 2003.

[3] D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. Technical report, University of Nijmegen, 2004. NIII Technical Report NIII-R0413.

[4] D. R. Cok and J. Kiniry. ESC/Java2 : Uniting ESC/Java and JML. Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an internet voting tally system. *Lecture Notes in Computer Science*, 3362:108–128, Jan. 2005.

[5] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, volume 37, 5 of *SIGPLAN*, pages 234–245, New York, June 2002. ACM Press.