Iterator Specification with Typestates

Kevin Bierhoff
Institute for Software Research
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
kevin.bierhoff @ cs.cmu.edu

ABSTRACT

Java iterators are notoriously hard to specify. This paper applies a general typestate specification technique that supports several forms of aliasing to the iterator problem. The presented specification conservatively captures iterator protocols and consistency rules. Two limitations of the specification are discussed.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specification—*Languages*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Design, languages, verification

Keywords

Iterator, typestate, specification, aliasing, verification

1. INTRODUCTION

The Java Collection API defines various rules for using iterators. It defines a *protocol* for accessing individual iterators. It also imposes restrictions on modifying iterated collections in order to keep iterators *consistent*. Similar rules are defined for C# enumerators.

Typestates augment the fixed type of a (mutable) object with a variable "condition" that describes the object's abstract state in its lifecycle [7]. A type system like Fugue's [4] that is based on this idea lets the programmer essentially define a state machine for each class. However, Fugue cannot fully capture iterator behavior due to its restrictions regarding aliasing and non-determinism.

This paper presents a specification of Java iterators based on a technique for typestate specifications in the presence of aliasing. The following section introduces some of the key concepts of this technique. The actual iterator specification is presented in section 3. Limitations of the specification are discussed in section 4 and section 5 concludes.

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

2. TYPESTATE SPECIFICATIONS

This section introduces a general technique for typestate specifications to the extent necessary for specifying iterators.

Hierarchical state spaces. We define orthogonal *state dimensions* with sets of mutually exclusive states [2]. The idea is to model separate aspects of object behavior separately. For example, we model Java iterators with three orthogonal dimensions (figure 1). At runtime, an iterator object will be in exactly one of the states from each dimension. The root state alive basically stands for "any state" and can be refined in an arbitrary number of dimensions. Similarly, a dimension stands for "any state" in that dimension.

States and dimensions are explicitly defined as part of an interface. For example, the next dimension depicted in figure 1 could be defined as follows.

```
states available, end refine alive as next
```

Dimensions or states do *not* correspond to implementation fields but information about fields can be *tied* to states, allowing implementation verification (see section 3.5).

Access permissions. Different variables could *alias* the same object and care must be taken to keep the "views" of those aliases onto the object consistent. Our approach is to associate variables with *access permissions* that are guaranteed to remain consistent.

A permission $perm(x, n, A)$ grants different levels of access to a part n of the state space (e.g., a state dimension) to a variable x. Permissions optionally carry additional information A about the exact state inside the part of the state space they cover (omitted otherwise). We use the following access levels.

- unique permissions guarantee that the variable is the only one that has access.
- full permissions guarantee that the variable is the only one that can change state.
- pure permissions give read-only access. There may be other pure permissions and at most one full permission around.

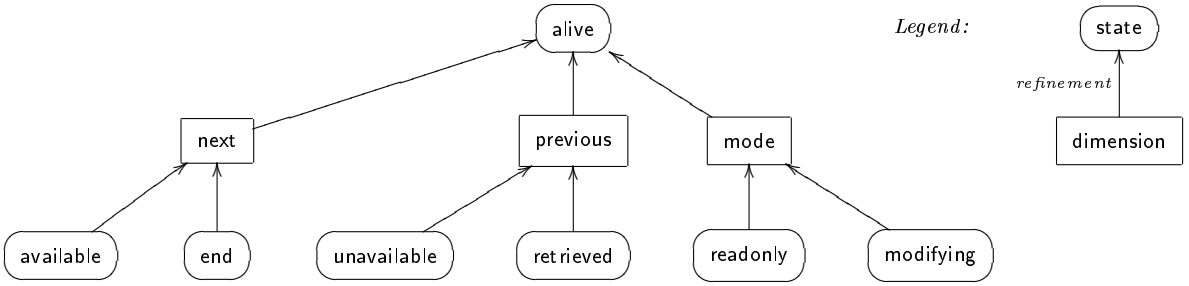

Figure 1: Iterator state space

As an example, pure(*this*, next, available) represents a pure permission on the next dimension of an (`Iterator`) receiver that is currently in the available state.

We use fractions [3] to keep track of splits. This lets us "collect" the full and all pure permissions and regain a unique permission. We omit fractions in specifications if they do not change (these permissions are universally quantified for all fractions).

Method specifications. Methods are specified with the decidable multiplicative-additive fragment of linear logic [5] (MALL). Pre- and post-conditions are separated with a linear implication ($\multimap$) and use conjunction ($\otimes$), internal choice (&), and external choice ($\oplus$). We include quantifiers for receiver *this* and return value *result* to make specifications self-sufficient. In one case we explicitly quantify over fractions.

The following example specifies the `hasNext` method for Java iterators. It requires a pure permission for the receiver's next dimension. The post-condition on the right-hand side of the implication is an external choice between conjunctions. The external choice indicates that the caller has no influence on whether `hasNext` will return `true` or `false`.

$$\forall this : \texttt{Iterator}.\ \exists result : \texttt{boolean}.$$
$$\mathsf{pure}(this, \mathsf{next}) \multimap (\mathsf{pure}(this, \mathsf{next}, \mathsf{end}) \otimes result = \texttt{false}) \oplus (\mathsf{pure}(this, \mathsf{next}, \mathsf{available}) \otimes result = \texttt{true})$$

The expressiveness of linear logic specifications is similar to our earlier work based on union and intersection types [2]. Tracking permissions with linear logic ensures that permissions cannot be duplicated. This is essential for sound static verification in a permission-based approach.

The notation used in this paper is fully explicit for clarity but we envision a more practical surface notation. In particular, quantifiers are implied by standard method signatures (see figure 2). Permissions could by default apply to the receiver or the position of a permission could imply which variable it applies to [2].

3. JAVA ITERATOR SPECIFICATION

This section presents an iterator specification using the techniques introduced in the last section. We state assumptions and goals before specifying iterators and iterables. Finally, we discuss how the specification can be used in verification.

3.1 Assumptions

This specification assumes single-threaded execution. We also assume that a unique permission is needed to modify a collection directly. This can be enforced with an appropriate specification of modifying methods in the `Collection` interface (which extends `Iterable`, specified below).

3.2 Specification Goals

Goals of the presented specification include the following.

- Allow creating an arbitrary number of iterators over collections ("iterables").
- Invalidate iterators before modification of the iterated collection.
- Capture the usage protocol of Java iterators.

3.3 Specifying Iterators

The `Iterator` specification is primarily concerned with capturing the protocol for using iterators (figure 2). In order to capture the expected usage of the `hasNext` and `next` methods we introduce a state dimension next with mutually exclusive states available and end. Calling `hasNext` conceptually performs a dynamic state test on this dimension: a `true` (`false`) return value corresponds with the available (end) state. A subsequent Boolean test on the return value allows a client to deduce the state of the iterator.

Notice that we do not change state with a call to `hasNext`, expressed by only requiring a pure permission for this call. Conversely, a call to `next` can potentially change the state of the next dimension and therefore needs a full permission to the receiver. It requires the next element to be available. Which of the two states in the next dimension will apply after the call is unknown. Thus our specification enforces the characteristic alternation of calls to `hasNext` and `next`.

The specification of `remove` requires two additional state dimensions. The mode dimension characterizes iterators as readonly or modifying. This dimension is *immutable* in the sense that an iterator cannot change between these states. The `remove` method can only be called on modifying iterators. Notice how the specification preserves that state like a side condition. With regard to the other dimension, `remove` prescribes that the previous element must be retrieved in order to `remove` it, making it also unavailable. Notice that the specification for `next` changes the previous dimension to retrieved. This enforces that `remove` can be called at most once after each call to `next`. (A newly created iterator will be in the unavailable state.)

3.4 Specifying Iterables

The `Iterable` interface is used to create iterators. We define two *cases* for this method. One case creates a read-only

```
interface Iterator<c : Iterable, g : alive → Fract> {
   states available, end refine alive as next
   states unavailable, retrieved refine alive as previous
   states readonly, modifying refine alive as mode

   boolean hasNext() :
      ∀this : Iterator. ∃result : boolean.
         pure(this, next) ⊸ (pure(this, next, available) ⊗ result = true)
                          ⊕ (pure(this, next, end) ⊗ result = false)

   Object next() :
      ∀this : Iterator. ∃result : Object.
         full(this, previous) ⊗ full(this, next, available) ⊸
                  full(this, previous, retrieved) ⊗ full(this, next) ⊗ pure(result, alive)

   void remove() :
      ∀this : Iterator.
         full(this, previous, retrieved) ⊗ pure(this, modifying) ⊸ full(this, previous, unavailable) ⊗ pure(this, modifying)

   void finalize() :
      ∀this : Iterator.
         (unique(this, alive, readonly) ⊸ pure(c, alive, g)) & (unique(this, alive, modifying) ⊸ full(c, alive, g))
}
```

Figure 2: `Iterator` interface specification

iterator and divides the fraction on the receiver's permission in half. The second half is given as a pure permission to the resulting readonly iterator. The other case requires a unique permission to the receiver in order to create a modifying iterator. Only a pure permission to the receiver is retained. Notice that our iterators are parameterized with a collection and a fraction (figure 2). These parameters help describing an iterator's permission to a collection.

Calling `iterator` with a full permission will always yield a read-only iterator (because only the first case applies). When calling it with a unique permission, on the other hand, both cases could apply. Thus `iterator` conceptually returns a readonly & modifying iterator, i.e., one of the two at the caller's choice, but not both. Another call to `iterator` forces readonly while calling `remove` on an iterator forces it to be modifying. As would be expected, retrieving elements from an iterator does not force one or the other. Thus we delay the choice between these two cases until it is inevitable. A full reference to an iterable indicates the existence of read-only iterators. A pure reference to an iterable indicates a modifying iterator.

3.5 Verification

Clients that use iterators as specified above can be verified by tracking linear permissions of bound variables. If reasoning about a decidable fragment of linear logic (MALL), dependently typed objects, and splitting and coalescing of permissions can be automated then verification can proceed automatically. Capabilities like Fugue's "state predicates" [4] let us reason about correctness of iterator *implementations* as well.

The last section described how specifications of `Iterable` and `Iterator` allow creating and using iterators in the right way. The question arises, how can a collection ever be modified again after an iterator was created? And how can we create a modifying iterator after other iterators were created?

A simple variable liveness analysis can determine when an iterator is no longer needed. If a variable dies that carries a unique permission to an iterator then the iterator becomes inaccessible and is subject to garbage collection. As soon as the iterator is dead we can get back its permission to the underlying collection.

We use the *finalizer* to specify this. A `finalize` method is defined for all Java objects and intended to be called in the process of garbage collection to release resources. We use it to release the permission to the iterated collection (figure 2; notice how the permission depends on the iterator's mode). Once released, the permission can be coalesced with any other permissions to the collection. Finalizing *all* created iterators restores the original unique permission to the collection, enabling direct modifications and creation of a modifying iterator.

4. LIMITATIONS AND COMPARISON

We identified the following two limitations of the specification presented here.

- The specification prevents the following legal use of Java iterators. A client can create two iterators and iterate over them in parallel until it decides to start modifying the collection through one of the iterators. This is legal if the other iterator is never used again. Our specification does not permit the modification because creating two iterators forces both to be read-only (see above) unless the second one is *created* after the first one dies. We are working on overcoming this problem by implicitly *changing* the iterator mode.
- As discussed above, the specification requires collections to be linear in order to be modified directly. This

```
interface Iterable {
   Iterator iterator() :
      ∀this : Iterable.
         (∀g : alive → Fract. ∃result : Iterator<this, g/2>.
            full(this, alive, g) ⊸ full(this, alive, g/2) ⊗ unique(result, alive, readonly))
       & (∃result : Iterator<this, alive ↦ 1/2>.
            unique(this, alive) ⊸ pure(this, alive, alive ↦ 1/2) ⊗ unique(result, alive, modifying))
}
```

Figure 3: `Iterable` interface specification

is stronger than one would expect; a full permission to the collection should suffice. The problem is that iterators expect collections to be immutable. We could model this with a state change of the collection (from "mutable" to "immutable"), but then we would need a dynamic test to know when the collection is mutable again. Instead we use fractions to count the number of iterators. Thus we trade states against aliasing restrictions and ease of use against flexibility in order to meet the Java specification (that does not include dynamic state tests on iterators).

The presented iterator specification uses a general technique that allows verification of iterator clients and implementations. We are aware of general techniques for functional specification (e.g. the JML [6], Spec# [1]) that rely on manual verification or automatic decision procedures but that are undecidable in general. Our technique supports certain forms of aliasing and is restricted to reasoning about typestates. The technique can capture many uses of iterators but we pay (modulo a cleverer specification) with the limitations mentioned above.

5. CONCLUSIONS

This paper presents a specification of Java iterators that may allow automatic verification of clients. The specification is conservative in that it respects the rules defined in the Java Collection API. To this end it limits aliasing of collections beyond what seems necessary and forbids a legal (albeit unusual) use of iterators.

6. ACKNOWLEDGMENTS

The author wishes to thank Ciera Christopher, Nels Beckman, and Jonathan Aldrich for helpful feedback on an earlier draft of this paper. This work was supported in part by NASA cooperative agreement NNA05CS30A, NSF grant CCF-0546550, and the Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems".

7. REFERENCES

[1] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004.

[2] K. Bierhoff and J. Aldrich. Lightweight object specification with typestates. In *ACM Symposium on the Foundations of Software Engineering*, pages 217–226, Sept. 2005.

[3] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.

[4] R. DeLine and M. Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*. Springer, 2004.

[5] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[6] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev28, Iowa State University, Department of Computer Science, July 2005.

[7] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12:157–171, 1986.