VC Generation for Functional Behavior and Non-Interference of Iterators

Bart Jacobs*
Dept. CS, K.U.Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium
bartj@cs.kuleuven.be

Frank Piessens
Dept. CS, K.U.Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium
frank@cs.kuleuven.be

Wolfram Schulte
Microsoft Research
One Microsoft Way
Redmond, WA, USA
schulte@microsoft.com

ABSTRACT
We propose a formalism for the full functional specification of enumerator methods, which are C# methods that return objects of type IEnumerable<T> or IEnumerator<T>. We further propose a sound modular automatic verification approach for enumerator methods implemented using C# 2.0’s iterator blocks (i.e., using yield return and yield break statements), and for client code that uses foreach loops. We require foreach loops to be annotated with special foreach loop invariants.

The approach prevents interference between iterator implementations and client code. Specifically, an enumerator method may read a field o.f only if o is reflexively-transitively owned by an object listed in the enumerator method’s reads clause, and the body of a foreach loop may not modify these objects. For example, we verify that a foreach loop iterating over an ArrayList does not modify the ArrayList. Note that one may break out of a foreach loop at any time to perform modifications before the iteration is complete. This in effect invalidates the iteration since the foreach loop cannot be resumed.

We support specification of non-deterministic enumerations, infinite enumerations, and enumerations that terminate with a checked exception, but not enumerations with side-effects. We support verification of an enumerator method only if it is implemented using yield statements, and verification of client code only if it performs a foreach loop on an enumerator method call. That is, the present approach does not support explicit creation or manipulation of IEnumerator<T> objects.

Our approach integrates easily with our concurrency approach (presented at ICFEM06), since both are based on read/write sets.

This approach was initially presented at FTIP05. Please refer to this paper for related work, references, and a soundness proof.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

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

Copyright 2006 ACM ISBN 1-59593-586-X/06/11 ...$5.00.

General Terms
Design, verification

Keywords
Formal specification, iterators, Boogie, verification

1. PROOF RULES

1.1 Spec#

We present our specification and verification method for the Iterator pattern in the context of the Spec# programming system, an extension of C# with preconditions, postconditions, non-null types, checked exceptions, loop invariants, object invariants, and other reliability features, and accompanied by a compiler that emits runtime checks and a static program verifier backed by an automatic theorem prover.

We hope to add support for our approach to the Spec# program verifier in the future.

The program verifier works by translating the Spec# source code into a guarded command program, which is then further translated into verification conditions that are passed to the theorem prover. The following guarded commands are relevant to this presentation:

- An assert C; statement causes an error to be reported if the condition C cannot be shown to always hold.
- An assume C; statement causes the verifier to consider only those program executions which either do not reach this statement or satisfy the condition C.
- A havoc x; statement assigns an arbitrary value to the variable x.

1.2 Specification of enumerator methods

In our formalism, methods are categorized as regular methods or enumerator methods. Enumerator methods must have a return type of IEnumerable<T> or IEnumerator<T>, for some T, and methods that have such return types are categorized as enumerator methods by default.

The syntax of an enumerator method’s contract differs from that of a regular method. In addition to requires and ensures clauses, an enumerator method may provide one or more invariant clauses, which declare the method’s enumeration invariants. Both the enumeration invariants and the ensures clauses may mention the keyword values, which denotes the sequence of elements yielded so far at a given point during the enumeration. The values keyword is of type Seq<T>, whose interface is given in Fig. 2. An enumeration invariant must hold at each point during an enumeration.
We propose a more concise syntax for simple cases like this one below.

These are “free of charge”, i.e. they provide assumptions but do not incur proof obligations, since they are guaranteed by the assert statements inserted at the yield return statements.

1.4 Verification of for-each loops

Our formalism supports proving rich properties of for-each loops by allowing their loop invariants to mention the keyword values, analogously with our approach to method contracts for enumerator methods. Here, too, the keyword is of type Seq<T>, where T is the element type of the enumeration, and represents the sequence of elements enumerated so far.

Here is an example of a client of our FromTo enumerator method:

```
int sum = 0;
foreach (int x in FromTo(1, 2))
    invariant sum += SeqTools.Sum(values);
assert sum == 3;
```

Now, consider a general for-each loop that uses a call of the general enumerator method M declared above as its enumerable expression:

```
foreach (T x in M(\vec{a})) invariant J; { S }
```

To verify this for-each loop, we translate it into the following for loop:

```
assume P[\vec{a}/\vec{p}]; Seq<T> values = new Seq<T>();
assert \forall \vec{a};
for (; ;)
    invariant I[\vec{a}/\vec{p}]; invariant J;
    {
        bool b; havoc b; if (~b) break;
        T x; havoc x;
        values.Add(x);
        assume I[\vec{a}/\vec{p}];
        assert J;
        S
    }
assert Q[\vec{a}/\vec{p}];
```

This means that for our example client, the program of Fig. 4 needs to be verified.

1.5 Exceptions

Our formalism supports the specification of enumerator methods that may throw checked exceptions, and the verification of the iterator methods that implement these. Enumerator methods may provide exceptional ensures clauses, and these may mention keyword values. An example is in Fig. 5.

---

1.3 Verification of iterator methods

We verify an enumerator method that is implemented as an iterator method (i.e., a method whose body is a C# 2.0 iterator block) by translating it into a guarded command program. Consider the following method:

```
IEnumerable<int> FromTo(int b, int e)
requires b ≤ e + 1;
invariant values.Count ≤ e + 1 − b;
invariant forall(int i in (0 : values.Count);
    values[i] == b + i);
ensures values.Count == e + 1 − b;
{
    for (int x = b; x ≤ e; x++)
        invariant values.Count == x − b;
        { yield return x; }
}
```

**Figure 1: Method FromTo**

```
public struct Seq<T> {
    public int Count { get; }
    public T this[int index]
        requires 0 ≤ index ∧ index < this.Count;
        { get; }
    public Seq();
    public void Add(T value);
        ensures this.Count == old(this).Count + 1;
        selects old(this).
        ensures forall(int i in (0 : old(this).Count);
            this[i] == old(old(this))[i];
        )
        ensures this[old(this)].Count == value;
}
```

**Figure 2: The Seq<T> type**

Fig. 1 shows an example of a method specified in our formalism.¹

![Image](image-url)

---

¹We propose a more concise syntax for simple cases like this one below.
int sum = 0;
assert 1 <= 2 + 1; Seq<int> values = new Seq<int>();
for (i; )
  invariant values.Count <= 2 + 1 - 1;
  invariant forall{ i in 0 : values.Count};
  values[i] == 1 + i;
  invariant sum == SeqTools.Sum(values);
  
  bool b; havoc b; if (!b) break;
T x; havoc x; values.Add(x);
assume values.Count <= 2 + 1 - 1;
assume forall{ i in 0 : values.Count};
  values[i] == 1 + i;
  sum += x;
} assume values.Count == 2 + 1 - 1; assert sum == 3;

Figure 4: Guarded command program generated as part of the verification of the example client

class OneElementException : CheckedException {}
class ThreeElementsException : CheckedException {}

IEnumerator<T> Baz()
ensures values.Count == 2;
throws OneElementException ensures values.Count == 1;
throws ThreeElementsException ensures values.Count == 3;

int n = 0;
try {
  foreach (int x in Baz()) invariant n == values.Count;
  { n++; } assert n == 2;
} catch (OneElementException) { assert n == 1;
} catch (ThreeElementsException) { assert n == 3; }

Figure 5: Enumerator methods that throw checked exceptions

[x = new C;] ≡
x = new C;
tid.W[x] = true;
tid.R[x] = 0;
x.inv = false;

[x = o.f;] ≡
assert tid.R'[o];
x = o.f;

[o.f = v;] ≡
assert tid.W'[o];
assert ~o.inv;
o.f = v;

[read (o) S] ≡
assert tid.R'[o];
assert o.inv;
tid.R[o]++;
foreach (p ∈ rep(o))
  tid.R[p]++;
[S]
foreach (p ∈ rep(o))
  tid.R[p]--;

[pack o;] ≡
assert tid.W'[o];
assert ~o.inv;
foreach (p ∈ rep(o))
  assert tid.W'[o];
  assert o.inv;
  }
foreach (p ∈ rep(o))
  tid.W[p] = false;
  o.inv = true;

[unpack o;] ≡
assert tid.W'[o];
assert o.inv;
foreach (p ∈ rep(o))
  tid.W[p] = true;
  o.inv = false;

[par (S₁, S₂);] ≡
let R = tid.par.R;
par ([S₁],
  tid.S₃[R] = R;
  [S₂]}
);

Figure 6: The programming methodology

1.6 Simplified alternative enumerator method
contract syntax

The general syntax presented above offers the flexibility of non-deterministic specifications; that is, it allows underspecification. Also, it allows a non-constructive description, as well as exceptional termination. However, often this flexibility is not needed, and for these cases we provide a simpler syntax, as follows:

IEnumerator<T> M(p)
requires P;
returns { i in (0:C); E);

For verification purposes, we expand this into the general syntax as follows:

IEnumerator<T> M(p)
requires P;
invariant values.Count <= C;
invariant forall{ i in 0 : values.Count};
  values[i] == E};
ensures values.Count == C;

2. AVOIDING INTERFERENCE

As is apparent from the explanations above, the implementation and the client of an enumerator method are verified as if they executed separately. However, they in fact execute in an interleaved fashion. To ensure soundness, our method prevents each party from observing side-effects of the execution of the other party.

Specifically, an enumerator method may not write fields of any pre-existing objects. Also, an enumerator method may declare in its contract a read set, using a reads clause, and it may only read fields of those pre-existing objects that are in its read set (or that are owned by such objects). Conversely, during enumeration, the client (i.e. the body of the for-each loop) may not write fields of these objects.

Here’s an example of an iterator pattern involving objects:

IEnumerator<int> EnumArray(int[]! a)
reads a; returns { i in (0:a.Length); a[i]};

{ for (int i = 0; i < a.Length; i++)
  invariant values.Count == i;
  { yield return a[i]; } }

int[] xs = {1, 2}; int sum = 0;
foreach (int x in EnumArray(xs))
  invariant sum == SeqTools.Sum(values);
  { sum ++ = x; } assert sum == 3;

The EnumArray method may read only the array, and the body of the for-each loop may not modify it. The declaration mark indicates that the argument for parameter a must not be null.

To statically and modularly verify the restrictions outlined above, our method for avoiding interference between the client and the implementation of an enumerator method requires that the program be written according to a programming methodology that is an extension of the Spec# object invariants methodology with support for read-only access. First, we briefly review the relevant aspects of the Spec# methodology. Then we present our extended version.

2.1 Spec# Methodology

In order to allow the object invariant for an object o to depend on objects other than o, Spec# introduces an ownership system; the
object invariant for \( o \) may depend on \( o \) and on any object transitively owned by \( o \). A program assigns ownership of an object \( p \) to \( o \) by writing \( p \) into a field of \( o \) declared rep while \( o \) is in the unpacked state, and then packing \( o \), which brings it into the packed state. The packed or unpacked state of an object is conceptually indicated by the value of a boolean field \( o.inv \), which is true if and only if \( o \) is in the packed state.

Packaging object \( o \) succeeds only if object \( p \) and the other objects pointed to by \( o \)'s rep fields are themselves already packed. Once \( o \) is packed, its owned objects may not be unpacked. Unpacking \( o \) again releases ownership of \( p \) and allows \( p \) to become owned by another object, or to become unpacked itself.

### 2.2 Programming Methodology

To understand the approach, it is useful to think of both parties in an enumeration as executing in separate threads. That is, the execution of a for-each statement starts the enumerating method in a new thread, executes the body of the for-each loop some number of times in the original thread, and then waits for the enumerating thread to finish. (We ignore for now the communication between both threads implied by the yielding of values, and the exact number of times the for-each loop is executed.) Note that we use the notion of threads as a reasoning tool only; we are not proposing implementing iterators using threads.

In our proposed system, each such thread \( t \) has a write set \( t.W \) and a read bag \( t.R \), both containing object references. The write set of a thread \( t \) contains those objects that were created by \( t \) and that are not currently committed to (i.e., owned by) some other object. The read bag of \( t \) contains an object \( o \) if \( t \) currently has read-only access to \( o \). The read bag is not a set, for technical reasons which will become clear later.

From \( t.W \) and \( t.R \), we derive the effective write set \( t.W' = t.W \cup t.R \) and the effective read set \( t.R' = t.W + t.R \). A thread \( t \) may read fields of any object in \( t.R' \), and it may write fields of any object in \( t.W' \), provided the object is unpacked.

The for-each statement may conceptually be thought of as being implemented in terms of a command par \((B_1, B_2)\); for parallel execution of two blocks \( B_1 \) and \( B_2 \). Execution of the par statement is complete only when execution of both blocks is finished. Suppose the par statement is being executed by a thread \( t_1 \). \( B_1 \) is executed in \( t_1 \), whereas \( B_2 \) is executed in a new thread, say \( t_2 \). The initial write set \( t_2.W \) of \( t_2 \) is the empty set, and the initial read bag is equal to that of \( t_1 \).

The proposed methodology is formally defined in Fig. 6, where tid denotes the current thread. The last rule translates a parallel execution statement by inserting an assertion that initializes the read bag of the newly created thread tid\(_{t_2}\) with the read bag of the creating thread tid\(_{t_1}\). The write set of the new thread remains initially empty. We use the following auxiliary definitions:

\[
\begin{align*}
t.W'[a] &\overset{\text{def}}{=} t.W[a] \land t.R[a] = 0 \\
&\quad t.R'[a] \overset{\text{def}}{=} t.W[a] \lor t.R[a] > 0 \\
&\quad \text{rep}(o) \overset{\text{def}}{=} \{ o.f | f \text{ is a rep field of } o \text{ and } o.f \neq \text{null} \}
\end{align*}
\]

The new read statement serves two purposes. Firstly, it allows a thread to take an object to which it has write access and make it read-only for the duration of the read statement, which enables it to be shared with newly created threads. Secondly, it allows a thread that has read access to an object \( o \) to gain access to \( o \)'s owned objects. That is, it replaces the unpack and pack operations if only read access is required. Note: in contrast to the unpack and pack pair, read blocks are re-entrant; that is, it is allowed to nest multiple read block executions on the same object. This is useful e.g. when writing recursive methods. This is also the reason why we need a read bag instead of a read set.

![Figure 7: Translation of the general for-each loop for the purpose of applying the non-interference methodology](image)

![Figure 8: Translation of the array example for the purpose of applying the non-interference methodology](image)