Iterators revisited: proof rules and implementation

Bart Jacobs[1], Erik Meijer[2], Frank Piessens[1], and Wolfram Schulte[2]

[1] Katholieke Universiteit Leuven, Dept. of Computer Science
Celestijnenlaan 200A, 3001 Leuven, Belgium
{bartj,frank}@cs.kuleuven.be
[2] Microsoft Research, Redmond, WA, USA
{emeijer,schulte}@microsoft.com

Abstract. The Iterator design pattern allows client code to access the elements of an aggregate object sequentially without exposing its underlying representation. Several modern programming languages, like Java or C#, include constructs that facilitate the implementation of the Iterator design pattern. They typically provide *for-each loops* for client code and *iterator methods* that specify the values yielded by the enumeration. However, when iterator methods are used to perform recursive enumerations, such as when enumerating the nodes of a tree, the worst-case running time often becomes quadratic.
We propose an extension of iterator methods called *nested iterator methods*, which simplify the coding of recursive enumerations, and which reduce the running time cost of such coding patterns to a linear function by retaining the recursion stack between iteration steps.
We also propose a new specification formalism and proof rules for *for-each* loops and iterator methods, including nested ones, suitable for implementation in automatic program verifiers such as ESCJava and Spec#. The core idea is the introduction of *enumeration invariants*, which must hold at each point during the iteration and which specify properties of the sequence of elements yielded so far. We also solve the problem of interference between client and enumerator, using an extension of the Spec# object invariants methodology with read-only access.

1 Introduction

Modern programming languages evolve. A heavily used part of the API or common programming pattern is a candidate for promotion to the host programming language. This also happens with the Iterator design pattern in C#.

In Version 1 of C# the *for-each* loop was introduced [9], which allows to retrieve the elements of an aggregate object sequentially without exposing its underlying representation. Here is a typical example of a **foreach** loop that iterates over a range of numbers to print them.

$$IEnumerable\langle\mathbf{int}\rangle\ xs = FromTo(1, 100);$$
$$\mathbf{foreach}\ (\mathbf{int}\ x\ \mathbf{in}\ xs)\ Console.WriteLine(x);$$

But providing the implementation for the involved $IEnumerable$ and $IEnumerator$ interfaces is cumbersome (see [9]) and that also hampered the verification of code that uses this pattern.

The complexity of implementing these interfaces was addressed in C# 2.0 [6]. It introduced *iterators*. Iterators are like ordinary methods except that they may *yield* multiple values instead of returning one value a single time. For example, the $FromTo$ method

$$\begin{array}{l} IEnumerable{<}\mathbf{int}{>}\ FromTo(\mathbf{int}\ b,\ \mathbf{int}\ e)\ \{ \\ \quad \mathbf{for}\ (\mathbf{int}\ x = b;\ x \leq e;\ x{+}{+})\ \mathbf{yield\ return}\ x; \\ \} \end{array}$$

yields, one by one, a finite, increasing stream of integers. Note that invoking such an iterator method does not immediately execute the iterator code, but rather immediately returns an instance of a compiler generated class implementing the $IEnumerable$ and $IEnumerator$ interface.

But C#'s iterators still have a caveat: they do not support nested iterators. But that's crucial when dealing with recursive data structures. The current iterators will for example have a quadratic behavior when they are used to return all text nodes from an Xml document, since they copy the stream produced by the recursive invocation.

$$\begin{array}{l} IEnumerable{<}XmlNode{>}\ Traverse(XmlNode\ n)\ \{ \\ \quad \mathbf{yield\ return}\ n; \\ \quad \mathbf{foreach}\ (XmlNode\ c\ \mathbf{in}\ n.ChildNodes) \\ \quad\quad \mathbf{foreach}\ (XmlNode\ m\ \mathbf{in}\ Traverse(c)) \\ \quad\quad\quad \mathbf{yield\ return}\ m; \\ \} \end{array}$$

Our newly proposed nested iterators, which follow the design of streams in Cω[4], simplify programming of recursive iterators, improve performance and as we will see later also facilitate verification. With nested iterators one can rewrite the example from above as follows.

$$\begin{array}{l} IEnumerable{<}XmlNode{>}\ Traverse(XmlNode\ n)\ \{ \\ \quad \mathbf{yield\ return}\ n; \\ \quad \mathbf{foreach}\ (XmlNode\ c\ \mathbf{in}\ n.ChildNodes) \\ \quad\quad \mathbf{yield\ foreach}\ Traverse(c); \\ \} \end{array}$$

Iterators are also provided in Spec#, which is a superset of the object-oriented .NET language C# 2.0, adding non-null types and code contracts [3]. This paper gives sound proof rules for specifying iterators. These rules are the basis for the Spec# compiler, which emits run-time checks to enforce the contracts of iterators, and the Spec# static program verifier, which attempts to prove iterators and *for-each* loops correct with respect to its contracts.

The contributions of this paper are thus as follows:

- We give for the first time proof rules for C# Version 2 *for-each* loops and iterators.
- We introduce nested iterators, their proof rules and fast implementation, which are expected to appear in one the next versions of C#.

Related work

Iterators have been introduced before (cf. [20, 13, 14, 8, 15, 11]), but they only recently made it into widely accepted languages like C# [9, 6] and Java [7]. Here we only cite work on formalizing and verifying iterators or that provided hints for a fast implementation of nested iterators.

Alphard, developed at CMU in the late 1970s [20], has two iterator consumption constructs: a *for* construct which is used for iteration over a data structure; and a variation of the for loop called a *first* loop which is used for search loops. Similar to C# version 1.0, Alphard iterators are defined by extending an enumeration form with two functions called *next* and *init*. Forms are not types, they serve as modules. It is possible to add verification information to an Alphard form, using an invariant clause, an initially clause, and pre and post conditions for each function. Proof obligations can then be given to ensure the correctness of the form: this means ensuring that invariants are maintained. Using the expansion of the *for* construct it is also possible to obtain a proof rule for the construct. However it is rather unwieldy. Also Alphard's proof rules haven't been proved sound nor implemented in a mechanical program verifier. This paper gives direct proof rules for iterator blocks and abstracts from their implementation.

Iterators in CLU, developed at MIT in the early 80's [13] were very restricted; see [15] for a detailed discussion of their shortcomings. But again CLU's iterators came already with proof rules [21]. The proposed technique for specifying CLU iterators involves adding assertions to two implicitly defined methods which follow the init and next pattern of Alphard. The assertions refer to state variables which can be decorated with subscripts pre and post, as well as a special iterator local state object (ie an auxiliary variable) first, which flags when the iterator is in the very first state, and history variables which remember values between invocations. The authors make explicit that their proposal is just a first step for the verification of iterators, they point out that the interaction of the iterator with the for loop needs further study.

Sather iterators or *iters* [15, 16] and loops are strictly more expressive than C# 2.0 iterators and *for-each* loops; that is, the latter correspond to a special case of the former. An iter may have a precondition and a postcondition. The postcondition must hold after each yield, but not after a quit. The iter contract syntax does not allow one to specify when the implementation is allowed to quit, which seems to make these contracts strictly less powerful than ours. Specifically, they would be incapable of verifying any of the examples of this paper. [16] describes optimizations for non-recursive iters. For recursive iters, they state the following "Our current implementation work is focussing on the optimization of recursive iters and some of the more complex loop structures. In appropriate circumstances, recursive iters can use more efficient stack structures than the direct coroutine implementation would." However, we are not aware of any further results of this work.

Cω, developed at Microsoft Research early 2000 [4], integrates data access into an object oriented host language. Streams, which are similar to our iterators, are used to virtualize data access to XML, SQL or even arbitrary object graphs. Cω provides a formal operational semantics and a formal type system for iterators. In fact Cω even proposes the use of nested iterators, but the study outlined in [4] is restricted to seman-

tical issues and doesn't provide any clue for efficient translations of nested iterators, which we address.

Grant Richins notes in his weblog [18] the fact that recursive iteration of the nodes of a tree using C# 2.0 iterators performs a number of object allocations that is linear in the number of nodes. To solve this, in a further posting [19] he eliminates the recursive invocation of iterators by keeping track of the current path using a stack data structure. This also removes the time penalty of this recursive iteration. We do not attempt to reduce object allocation.

Iterators have caused problems in the research on ownership systems (cf. [5, 1, 2]). Typically, the dilemma comes up of whether the collection should own/contain the iterator or vice versa. Some systems provide solutions that allow iterators to be programmed, but they do not address the problem of verifying their functional behavior. We envisage as a promising approach allowing iterators to have shared ownership of the collections by temporarily making them read-only.

Finally, one should mention that iterators have also been studied in the context of the refinement calculus [10]. However their formal analysis isn't helpful for us. They translate iterators into catamorphisms, while we are interested in generating verification conditions and fast implementation techniques.

2 Proof rules

2.1 Spec#

We present our specification and verification method for the Iterator pattern in the context of the Spec# programming system [3, 17], an extension of C# with preconditions, postconditions, non-null types, checked exceptions [12], loop invariants, object invariants [2], and other reliability features, and accompanied by a compiler that emits runtime checks and a static program verifier backed by an automatic theorem prover.

The program verifier works by translating the Spec# source code into a guarded commands 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 $\mathbf{assert}\ C$; statement causes an error to be reported if the condition C cannot be shown to always hold.
- An $\mathbf{assume}\ C$; statement causes the verifier to consider only those program executions which either do not reach this statement or satisfy the condition C. In particular, $\mathbf{assume\ false}$ causes the program verifier to consider an execution that reaches this statement without errors to be a valid execution, regardless of any statements that may follow this statement.
- A $\mathbf{havoc}\ x$; statement assigns an arbitrary value to the variable x.

2.2 Specification of enumerator methods

In the most general form of our proposed formalism, the specification of method $FromTo$ is as in Fig. 1. [3]

[3] We propose a more concise syntax for simple cases like this one below.

```
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; }
}
```

Fig. 1. Method *FromTo*

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.

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

Fig. 2. The $Seq<T>$ type

2.3 Verification of iterator methods

We verify an enumerator method that is implemented as an iterator method by translating it into a guarded commands program. Consider the following method:

$$\mathit{IEnumerable}\langle T\rangle\ M(\boldsymbol{p})\ \mathbf{requires}\ P;\ \mathbf{invariant}\ I;\ \mathbf{ensures}\ Q;\ \{\ B\ \}$$

It gets translated into the following:

$$\mathbf{assume}\ P;\ \mathit{Seq}\langle T\rangle\ \mathit{values} = \mathbf{new}\ \mathit{Seq}\langle T\rangle();\ \mathbf{assert}\ I;\ [\![B]\!]\ \mathbf{assert}\ Q;$$

where

$$[\![\mathbf{yield\ return}\ v;]\!] \equiv\ \mathit{values}.\mathit{Add}(v);\ \mathbf{assert}\ I;$$
$$[\![\mathbf{yield\ break};]\!] \equiv\ \mathbf{assert}\ Q;\ \mathbf{assume\ false};$$

That is, we verify that the enumeration invariants hold for the empty sequence, as well as after each **yield return** operation. Also, we check the postcondition at each **yield break** operation.

As a convenience, we insert I as a loop invariant into each loop in B.[4]

Applied to our *FromTo* example from Fig. 1, this yields the following:

$$\begin{array}{l}
\mathbf{assume}\ b \le e + 1;\\
\mathit{Seq}\langle \mathit{int}\rangle\ \mathit{values} = \mathbf{new}\ \mathit{Seq}\langle \mathit{int}\rangle();\\
\mathbf{assert}\ \mathit{values}.\mathit{Count} \le e + 1 - b;\\
\mathbf{assert\ forall}\{\mathbf{int}\ i\ \mathbf{in}\ (0 : \mathit{values}.\mathit{Count});\ \mathit{values}[i] == b + i\};\\
\mathbf{for}\ (\mathbf{int}\ x = b;\ x \le e;\ x{+}{+})\\
\quad \mathbf{invariant}\ \mathit{values}.\mathit{Count} \le e + 1 - b;\\
\quad \mathbf{invariant\ forall}\{\mathbf{int}\ i\ \mathbf{in}\ (0 : \mathit{values}.\mathit{Count});\ \mathit{values}[i] == b + i\};\\
\quad \mathbf{invariant}\ \mathit{values}.\mathit{Count} == x - b;\\
\{\\
\quad \mathit{values}.\mathit{Add}(x);\\
\quad \mathbf{assert}\ \mathit{values}.\mathit{Count} \le e + 1 - b;\\
\quad \mathbf{assert\ forall}\{\mathbf{int}\ i\ \mathbf{in}\ (0 : \mathit{values}.\mathit{Count});\ \mathit{values}[i] == b + i\};\\
\}\\
\mathbf{assert}\ \mathbf{values}.\mathit{Count} == e + 1 - b;
\end{array}$$

2.4 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 $\mathit{Seq}\langle T\rangle$, where T is the element type of the enumeration, and represents the sequence of elements enumerated so far. [5]

[4] 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.

[5] The **values** keyword's scope includes all specification expressions in the body of the *for-each* loop. However, a nested *for-each* loop hides the outer loop's **values** keyword. Still, one can give the inner loop access to this information by assigning the **values** keyword to an auxiliary logical variable in the body of the outer loop.

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

$$
\begin{array}{l}
\textbf{int } sum = 0; \\
\textbf{foreach } (\textbf{int } x \textbf{ in } FromTo(1, 2)) \\
\quad \textbf{invariant } sum == SeqTools.Sum(\textbf{values}); \\
\{\ sum \mathrel{+}= x;\ \} \\
\textbf{assert } sum == 3;
\end{array}
$$

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

$$\textbf{foreach } (T\ x \textbf{ in } M(\boldsymbol{a})) \textbf{ invariant } J;\ \{\ S\ \}$$

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

$$
\begin{array}{l}
\textbf{assert } P[\boldsymbol{a}/\boldsymbol{p}];\ Seq\langle T\rangle\ values = \textbf{new } Seq\langle T\rangle(); \\
\textbf{for } (;;) \\
\quad \textbf{invariant } I[\boldsymbol{a}/\boldsymbol{p}];\ \textbf{invariant } J; \\
\{ \\
\quad \textbf{bool } b;\ \textbf{havoc } b;\ \textbf{if } (\neg b)\ \textbf{break};\ T\ x;\ \textbf{havoc } x;\ values.Add(x); \\
\quad \textbf{assume } I[\boldsymbol{a}/\boldsymbol{p}]; \\
\quad S \\
\} \\
\textbf{assume } Q[\boldsymbol{a}/\boldsymbol{p}];
\end{array}
$$

This means that for our example client, the following needs to be verified:

$$
\begin{array}{l}
\textbf{int } sum = 0; \\
\textbf{assert } 1 \leq 2 + 1;\ Seq\langle\textbf{int}\rangle\ values = \textbf{new } Seq\langle\textbf{int}\rangle(); \\
\textbf{for } (;;) \\
\quad \textbf{invariant } values.Count \leq 2 + 1 - 1; \\
\quad \textbf{invariant forall}\{\textbf{int } i \textbf{ in } (0 : \textbf{values}.Count);\ \textbf{values}[i] == 1 + i\}; \\
\quad \textbf{invariant } sum == SeqTools.Sum(values); \\
\{ \\
\quad \textbf{bool } b;\ \textbf{havoc } b;\ \textbf{if } (\neg b)\ \textbf{break}; \\
\quad T\ x;\ \textbf{havoc } x;\ values.Add(x); \\
\quad \textbf{assume } values.Count \leq 2 + 1 - 1; \\
\quad \textbf{assume forall}\{\textbf{int } i \textbf{ in } (0 : \textbf{values}.Count);\ \textbf{values}[i] == 1 + i\}; \\
\quad sum \mathrel{+}= x; \\
\} \\
\textbf{assume } values.Count == 2 + 1 - 1;\ \textbf{assert } sum == 3;
\end{array}
$$

2.5 Verification of nested iterator methods

Nested iterator methods are iterator methods that contain **yield foreach** statements. Their translation is the same as for general iterator methods, with the following addi-

tional rule:

$$
\begin{array}{l}
[\![\textbf{yield foreach } M'(\boldsymbol{a}');]\!] \equiv \\
\quad \textbf{assert } P'[\boldsymbol{a}'/\boldsymbol{p}']; \\
\quad \mathit{Seq}\!<\!T\!>\ \mathit{values}'; \ \textbf{havoc } \mathit{values}'; \\
\quad \textbf{assume } I'[\boldsymbol{a}'/\boldsymbol{p}', \mathit{values}'/\mathit{values}]; \ \textbf{assume } Q'[\boldsymbol{a}'/\boldsymbol{p}', \mathit{values}'/\mathit{values}]; \\
\quad \mathit{values}.\mathit{AddRange}(\mathit{values}'); \\
\quad \textbf{assert } I;
\end{array}
$$

That is, the enumeration invariant I of the caller enumeration is checked under the assumption that a sequence of elements was yielded that satisfies the callee's enumeration invariant, as well as its postcondition.

2.6 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**.

For example:

$$
\begin{array}{l}
\textbf{class } \mathit{OneElementException} \ : \ \mathit{CheckedException} \ \{\} \\
\textbf{class } \mathit{ThreeElementsException} \ : \ \mathit{CheckedException} \ \{\} \\
\\
\mathit{IEnumerable}\!<\!\textbf{int}\!>\ \mathit{Baz}() \\
\quad \textbf{ensures } \mathit{values}.\mathit{Count} == 2; \\
\quad \textbf{throws } \mathit{OneElementException} \ \textbf{ensures } \mathit{values}.\mathit{Count} == 1; \\
\quad \textbf{throws } \mathit{ThreeElementException} \ \textbf{ensures } \mathit{values}.\mathit{Count} == 3; \\
\\
\textbf{int } n = 0; \\
\textbf{try } \{ \\
\quad \textbf{foreach } (\textbf{int } x \textbf{ in } \mathit{Baz}()) \ \textbf{invariant } n == \mathit{values}.\mathit{Count}; \ \{ \ n{+}{+}; \ \} \\
\quad \textbf{assert } n == 2; \\
\} \\
\textbf{catch } (\mathit{OneElementException}) \ \{ \ \textbf{assert } n == 1; \ \} \\
\textbf{catch } (\mathit{ThreeElementException}) \ \{ \ \textbf{assert } n == 3; \ \}
\end{array}
$$

2.7 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:

$$
\mathit{IEnumerable}\!<\!T\!>\ M(\boldsymbol{p}) \ \textbf{requires } P; \ \textbf{returns } \{\textbf{int } i \textbf{ in } (0{:}C); E\};
$$

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

$$
\begin{array}{l}
\mathit{IEnumerable}\langle T\rangle\ M(\boldsymbol{p}) \\
\quad \textbf{requires}\ P; \\
\quad \textbf{invariant}\ \textbf{values}.\mathit{Count} \leq C; \\
\quad \textbf{invariant}\ \textbf{forall}\{\textbf{int}\ i\ \textbf{in}\ (0{:}\textbf{values}.\mathit{Count});\ \textbf{values}[i] == E\}; \\
\quad \textbf{ensures}\ \textbf{values}.\mathit{Count} == C;
\end{array}
$$

3 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, as will become clear in the next section. 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 the 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:

$$
\begin{array}{l}
\mathit{IEnumerable}\langle\textbf{int}\rangle\ \mathit{EnumArray}(\textbf{int}[]!\ a) \\
\quad \textbf{reads}\ a;\ \textbf{returns}\ \{\textbf{int}\ i\ \textbf{in}\ (0{:}a.\mathit{Length});\ a[i]\}; \\
\{ \\
\quad \textbf{for}\ (\textbf{int}\ i = 0;\ i < a.\mathit{Length};\ i{+}{+})\ \textbf{invariant}\ \textbf{values}.\mathit{Count} == i; \\
\quad \{\ \textbf{yield return}\ a[i];\ \} \\
\} \\
\\
\textbf{int}[]\ xs = \{1, 2\};\ \textbf{int}\ \mathit{sum} = 0; \\
\textbf{foreach}\ (\textbf{int}\ x\ \textbf{in}\ \mathit{EnumArray}(xs)) \\
\quad \textbf{invariant}\ \mathit{sum} == \mathit{SeqTools}.\mathit{Sum}(\textbf{values}); \\
\{\ \mathit{sum} \mathrel{+}= x;\ \} \\
\textbf{assert}\ \mathit{sum} == 3;
\end{array}
$$

The *EnumArray* method may read only the array, and the body of the **foreach** loop may not modify it. The exclamation 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 [2] with support for read-only access. First, we briefly review the relevant aspects of the Spec# methodology. Then we present our extended version.

3.1 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.

Packing 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.

3.2 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 enumerator 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 enumerator 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 object 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* $t.W' = t.W - t.R$ 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 $\mathbf{par}\ (B_1,\ B_2)$; 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. 3, where **tid** denotes the current thread. The last rule translates a parallel execution statement by inserting an assignment that initializes the read bag of the newly created thread $\mathbf{tid}_{S_2}$ with the read bag of the creating thread $\mathbf{tid}_{\mathbf{par}}$. The write set of the new thread remains initially empty.

We use the following auxiliary definitions:

$$t.W'[o] \stackrel{\text{def}}{=} t.W[o] \wedge t.R[o] = 0 \qquad t.R'[o] \stackrel{\text{def}}{=} t.W[o] \vee t.R[o] > 0$$

$$\mathrm{rep}(o) \stackrel{\text{def}}{=} \{o.f \mid f \text{ is a } \mathbf{rep} \text{ field of } o \text{ and } o.f \neq \mathrm{null}\}$$

$[\![x = \mathbf{new}\ C;]\!] \equiv$
  $x = \mathbf{new}\ C;$
  $\mathbf{tid}.W[x] = \mathbf{true};$
  $\mathbf{tid}.R[x] = 0;$
  $x.inv = \mathbf{false};$

$[\![x = o.f;]\!] \equiv$
  $\mathbf{assert}\ \mathbf{tid}.R'[o];$
  $x = o.f;$

$[\![o.f = v;]\!] \equiv$
  $\mathbf{assert}\ \mathbf{tid}.W'[o];$
  $\mathbf{assert}\ \neg o.inv;$
  $o.f = v;$

$[\![\mathbf{pack}\ o;]\!] \equiv$
  $\mathbf{assert}\ \mathbf{tid}.W'[o];$
  $\mathbf{assert}\ \neg o.inv;$
  $\mathbf{foreach}\ (p \in \mathrm{rep}(o))\ \{$
    $\mathbf{assert}\ \mathbf{tid}.W'[o];$
    $\mathbf{assert}\ o.inv;$
  $\}$
  $\mathbf{foreach}\ (p \in \mathrm{rep}(o))$
    $\mathbf{tid}.W[p] = \mathbf{false};$
  $o.inv = \mathbf{true};$

$[\![\mathbf{unpack}\ o;]\!] \equiv$
  $\mathbf{assert}\ \mathbf{tid}.W'[o];$
  $\mathbf{assert}\ o.inv;$
  $\mathbf{foreach}\ (p \in \mathrm{rep}(o))$
    $\mathbf{tid}.W[p] = \mathbf{true};$
  $o.inv = \mathbf{false};$

$[\![\mathbf{read}\ (o)\ S]\!] \equiv$
  $\mathbf{assert}\ \mathbf{tid}.R'[o];$
  $\mathbf{assert}\ o.inv;$
  $\mathbf{tid}.R[o]{+}{+};$
  $\mathbf{foreach}\ (p \in \mathrm{rep}(o))$
    $\mathbf{tid}.R[p]{+}{+};$
  $[\![S]\!]$
  $\mathbf{foreach}\ (p \in \mathrm{rep}(o))$
    $\mathbf{tid}.R[p]{-}{-};$
  $\mathbf{tid}.R[o]{-}{-};$

$[\![\mathbf{par}\ (S_1,\ S_2);]\!] \equiv$
  $\mathbf{let}\ R = \mathbf{tid}_{\mathbf{par}}.R;$
  $\mathbf{par}\ ([\![S_1]\!],\ \{$
    $\mathbf{tid}_{S_2}.R = R;$
    $[\![S_2]\!]$
  $\});$

Fig. 3. The programming methodology

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*.

Consider the general *for-each* statement shown in Section 2.4. For the purpose of applying the proposed methodology, it is equivalent with the program in Fig. 4, assuming that method M has a **reads** R; clause.

For the array example above, this yields the program in Fig. 5.

3.3 Yielding of references

If the values yielded by an enumeration are or contain object references, then this begs the question as to what access both parties have to these objects. In the basic methodology proposed in this paper, there is no change in the access that both parties have at the time of the yield operation. This is often the desired semantics; for example when yielding the elements of a collection, the elements are typically owned by the client and should remain so. However, as future work, we envisage supporting the specification and verification of enumerator methods that e.g. create new objects and then when yielding the objects transfer ownership (i.e. write access) to the client, or that share read access to these objects between both parties. The read-write-access methodology

```
assert P[a/p];
read (R) {
  par (| {
    Seq<T> values = new Seq<T>();
    for (;;) invariant I[a/p]; invariant J;
    {
      bool b; havoc b; if (¬b) break;
      T x; havoc x; values.Add(x); assume I[a/p];
      S
    }
    assume Q[a/p];
  }, { Seq<T> values = new Seq<T>(); assert I; [[B]] assert Q; });
}
```

Fig. 4. Translation of the general *for-each* loop for the purpose of applying the non-interference methodology

already supports this, but it is future work to determine how this should be specified in the enumerator method contract or what the impact on the verification of Iterator patterns is.

4 Implementation

This section analyzes the current implementation technique of iterators and shows how it can naturally be extended to deal with nested iterators, too.

4.1 Translation of *for-each* loops

To understand the behavior of enumerations in detail, let's look at their implementations as exemplified in the C# Language Specification [9]. It says (slightly simplified) that the following *for-each* loop

$$\textbf{foreach}\ (T\ x\ \textbf{in}\ C)\ S$$

abbreviates the following **while** loop: [6]

$$IEnumerable\langle T\rangle\ c = C;\ IEnumerator\langle T\rangle\ e = c.GetEnumerator();$$
$$\textbf{while}\ (e.MoveNext())\ \{\ T\ x = e.Current;\ S\ \}$$

In this expansion the $GetEnumerator$ method returns an enumerator that enumerates the elements of the collection. The **while** loop first calls $MoveNext$ to advance the

[6] The .NET Framework's variant of the Iterator design pattern, as embodied in the $IEnumerator\langle T\rangle$ and related interfaces, prescribes that a client must signal to the enumerator implementation that resources allocated by the enumerator implementation may be disposed of, by calling a $Dispose$ method on the enumerator object. The **foreach** statement performs this call implicitly. We ignore this detail in this paper because it is inconsequential to our discussion.

```
int[] xs = {1, 2}; int sum = 0;
read (xs)
{
  par ({
    Seq<T> values = new Seq<T>();
    for (; ; )
      invariant values.Count ≤ xs.Length;
      invariant forall{int i in (0:values.Count); values[i] == xs[i]};
      invariant sum == SeqTools.Sum(values);
    {
      bool b; havoc b; if (¬b) break; T x; havoc x; values.Add(x);
      assume values.Count ≤ xs.Length;
      assume forall{int i in (0:values.Count); values[i] == xs[i]};
      sum += x;
    }
    assume values.Count == xs.Length;
  }, {
    Seq<T> values = new Seq<T>();
    assert values.Count ≤ xs.Length;
    assert forall{int i in (0:values.Count); values[i] == xs[i]};
    for (int i = 0; i < xs.Length; i++)
      invariant values.Count ≤ xs.Length;
      invariant forall{int i in (0:values.Count); values[i] == xs[i]};
      invariant values.Count == i;
    {
      values.Add(xs[i]);
      assert values.Count ≤ xs.Length;
      assert forall{int i in (0:values.Count); values[i] == xs[i]};
    }
    assert values.Count == xs.Length;
  });
}
assert sum == 3;
```

Fig. 5. Translation of the array example for the purpose of applying the non-interference methodology

enumerator to the first element of the collection before reading the value of $Current$. After the end of the collection is passed, the enumerator is positioned after the last element in the collection, and calling $MoveNext$ returns **false**.

4.2 Translation of non-nested iterators

First, let's consider a translation of iterator methods into plain C#. This is essentially the example translation described in the C# 2.0 Language Specification. [7] As an example, consider the translation in Fig. 6 of the $FromTo$ method introduced earlier.

4.3 Translation of nested iterators

The translation of nested iterators is very similar to the translation of non-nested iterators. We illustrate the translation using the following recursive version of method $FromTo$. The translation is in Figs. 7, 8, and 9.[8]

$$
\begin{array}{l}
\textbf{static } \mathit{IEnumerable}\langle\textbf{int}\rangle\ \mathit{FromToNested}(\textbf{int } b,\ \textbf{int } e)\ \{ \\
\quad \textbf{if } (b > e)\ \textbf{yield break}; \\
\quad \textbf{yield return } b;\ \textbf{yield foreach } \mathit{FromToNested}(b+1,\ e); \\
\}
\end{array}
$$

The main difference in the translation is that the object returned by the translation of $FromToNested$ now derives from an abstract class $NestedEnumerable\langle\textbf{int}\rangle$, and that the enumerator objects derive from an abstract class $NestedEnumerator\langle\textbf{int}\rangle$.

The idea behind the $NestedEnumerator\langle T\rangle$ class is that it generalizes the $IEnumerator\langle T\rangle$ pattern from a linear list to an n-ary tree. But before looking in detail at the code for $NestedEnumerator\langle T\rangle$, let's look at the state machine of $NestedMoveNext$, which resulted from the translation of the $FromToNested$ body.
The $NestedMoveNext$ method returns one of four values: it returns either $Done$ indicating that the enumeration is finished, $Value$ indicating that the iterator produced a single value, $Enumerator$ indicating that the iterator produced a nested enumeration, or $TailEnumerator$ indicating that the iterator produced a nested enumeration and that it is now done.

The abstract base classes $NestedEnumerable$ and $NestedEnumerator$ are similar to the interfaces $IEnumerable$ and $IEnumerator$, except that they gather some reusable members.

The most interesting class is $RootEnumerator$. It implements the $IEnumerator$ interface based on a given $NestedEnumerator$ object. Its $MoveNext$ method operates on a stack of nested enumerators. First, $MoveNext$ asks the $NestedMoveNext$ method for the next information. If it is a value, it can be returned as usual. If it is a nested iterator,

[7] For increased efficiency, the C# compiler emits a single class that implements both the $IEnumerable\langle T\rangle$ and $IEnumerator\langle T\rangle$ interfaces. To simplify the exposition, we use separate classes. This is inconsequential for our purposes.

[8] Again, we can merge the $FromToNestedEnumerable$ and the $FromToNestedEnumerator$ classes into one, but for clarity of exposition we use separate classes.

```
static IEnumerable<int> FromTo(int b, int e) {
  return new FromToEnumerable(b, e);
}

class FromToEnumerable : IEnumerable<int> {
  int b, e;
  public FromToEnumerable(int _b, int _e) { b = _b; e = _e; }
  public IEnumerator<int> GetEnumerator() {
    return new FromToEnumerator(b, e);
  }
}

class FromToEnumerator : IEnumerator<int> {
  int b, e, pc, current, i;
  public FromToEnumerator(int _b, int _e) { b = _b; e = _e; }
  public int Current {
    get {
      if (pc == 0 ∨ pc == 3)
        throw new InvalidOperationException();
      return current;
    }
  }
  public bool MoveNext() {
    switch (pc) {
      case 0: i = b; goto case 1;
      case 1:
        if (i > e) goto case 3;
        current = i; pc = 2; return true;
      case 2: i++; goto case 1;
      case 3: pc = 3; return false;
    }
  }
}
```

Fig. 6. Translation of *FromTo*

```
public abstract class NestedEnumerable<T> : IEnumerable<T> {
  public abstract NestedEnumerator<T> GetNestedEnumerator();
  public IEnumerable<T> GetEnumerator() {
    return new RootEnumerator(this.GetNestedEnumerator());
  }
}
public enum NestedEnumState { Value, Enumerator, Done, TailEnumerator }
public abstract class NestedEnumerator<T> {
  protected int pc;
  protected T currentValue;
  protected NestedEnumerator<T> currentEnumerator;
  public abstract NestedEnumState NestedMoveNext();
  public static NestedEnumerator<T> GetNestedEnumerator(IEnumerable<T> c) {
    NestedEnumerable<T> nc = c as NestedEnumerable<T>;
    return nc == null ? new EnumeratorAdaptor(c) : nc.GetNestedEnumerator();
  }
}
```

Fig. 7. Support classes for the translation of nested iterator methods (*part 1*)

it is placed on the stack. And if the top enumeration is exhausted, the stack is popped and processing continues. If the stack is empty, the root enumeration is complete.

When performing a nested enumeration using non-nested iterators, execution passes through each enclosing enumeration whenever the client calls the $MoveNext$ method on the root enumerator. That is, the recursion stack is built up and torn down on the call stack at each iteration step. By retaining the recursion stack in a data structure, nested iterators avoid this overhead, which is linear in the depth of recursion, multiplied by the number of iteration steps. For example, for a recursively enumerated linked list, the number of calls is quadratic in the number of elements, and for a recursively enumerated balanced tree of n nodes, the number of calls is $O(n\log(n))$. When using nested iterators, in each of these cases, the number of calls, as well as the overall time complexity, is $O(n)$.

Figure 10 shows the performance difference. We compute ranges once using a C# version 2 non-nested but recursive version of the $FromTo$ iterator and compare it with a nested recursive iterator (specifically, the $FromToNested$ method). In each case we generate 500 numbers. We repeat the experiment 100 times. In each case we observe that the current implementation has quadratic time behavior whereas the proposed implementation is linear. But since we often experienced performance degradation sparks, Fig 10 doesn't show the average behavior, but two particular runs.

5 Conclusion

Iterators are omnipresent and thus it is time to consider them as first class citizens in our languages: both for verification and for efficient execution. We gave for the first time straightforward proof rules for iterators that respect their laziness and that work

```
public class RootEnumerator<T> : IEnumerator<T> {
  Stack<NestedEnumerator<T>> stack = new Stack<NestedEnumerator<T>>();
  public RootEnumerator(NestedEnumerator<T> e) {
    stack.Push(e);
  }
  public T Current {
    get {
      if (stack.Count == 0 ∨ stack.Peek().pc == 0)
        throw new InvalidOperationException();
      return stack.Peek().currentValue;
    }
  }
  public bool MoveNext() {
    while (true) {
      if (stack.Count == 0) return false;
      switch (stack.Peek().NestedMoveNext()) {
        case NestedEnumState.Value:
          return true;
        case NestedEnumState.Enumerator:
          stack.Push(stack.Peek().currentEnumerator);
          break;
        case NestedEnumState.Done:
          stack.Pop();
          break;
        case NestedEnumState.TailEnumerator:
          NestedEnumerator<T> e = stack.Peek().currentEnumerator;
          stack.Pop(); stack.Push(e);
          break;
      }
    }
  }
}
public class EnumeratorAdaptor<T> : NestedEnumerator<T> {
  IEnumerator<T> e;
  public EnumeratorAdaptor(IEnumerable<T> c) {
    e = c.GetEnumerator();
  }
  public NestedEnumState NestedMoveNext() {
    if (e.MoveNext())
      { currentValue = e.Current; return NestedEnumState.Value; }
    else
      return NestedEnumState.Done;
  }
}
```

Fig. 8. Support classes for the translation of nested iterator methods (*part 2*)

```
static IEnumerable<T> FromToNested(int a, int b)
{ return new FromToNestedEnumerable(a, b); }
class FromToNestedEnumerable : NestedEnumerable<int> {
  int b, e;
  public FromToNestedEnumerable(int _b, int _e) { b = _b; e = _e; }
  public override NestedEnumerator<int> GetNestedEnumerator() {
    return new FromToNestedEnumerator(b, e);
  } }
public class FromToNestedEnumerator : NestedEnumerator<int> {
  int b, e;
  public FromToNestedEnumerator(int _b, int _e) { b = _b; e = _e; }
  public override NestedEnumState NestedMoveNext() {
    switch (pc) {
      case 0 : if (b > e) goto case 2;
        currentValue = b; currentEnumerator = null;
        pc = 1; return NestedEnumState.Value;
      case 1 : currentValue = default(T);
        currentEnumerator = GetNestedEnumerator(FromToNested(b + 1, e));
        pc = 2; return NestedEnumState.TailEnumerator;
      case 2 : pc = 2; return NestedEnumState.Done;
    } } }
```

Fig. 9. Translation of *FromToNested*

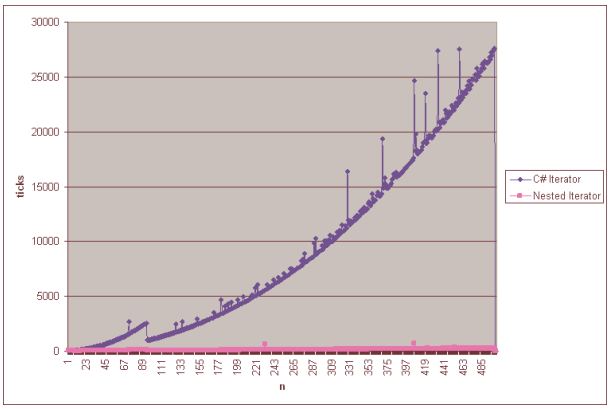

Fig. 10. Running time with non-nested iterators (*upper line*) and nested iterators (*lower line*)

particularly well with their use in *for-each* loops. We also proposed the introduction of nested iterators. In contrast with existing implementations, which lack nested iterators, our implementation has linear time complexity, which makes nested iterators particular amenable for iterating over recursive data structures.

Acknowledgements

The authors would like to thank the members of the Spec# team at Microsoft Research and Jan Smans at K.U.Leuven for their helpful comments.

Bart Jacobs co-authored this paper during an internship at Microsoft Research. Bart Jacobs is a Research Assistant of the Research Fund - Flanders (F.W.O.-Vlaanderen).

References

1. Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP*, 2004.
2. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
3. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS*, Lecture Notes in Computer Science. Springer, 2004.
4. Gavin Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in cω. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, Lecture Notes in Computer Science. Springer, July 2005.
5. Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *POPL*, 2003.
6. Microsoft Corporation. C# Language Working Draft 2.7, June 2004 (PDF). URL: http://msdn.microsoft.com/vcsharp/programming/language/, 2004.
7. Sun Corporation. JSR 201: Extending the JavaTM Programming Language with Enumerations, Autoboxing, Enhanced for loops and Static Import. URL: http://jcp.org/en/jsr/detail?id=201, 2004.
8. R. Griswold and M. Griswold. *The Icon programming language*. Prentice Hall, 1990.
9. Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. Addison-Wesley, 2003.
10. Steve King and Carroll Morgan. An iterator construct for the refinement calculus. In *Fourth Irish Workshop on Formal Methods*, 2000.
11. A. Krall and J. Vitek. On extending Java. In *Proceedings of JMLC*, 1997.
12. K. Rustan M. Leino and Wolfram Schulte. Exception safety for C#. In *SEFM 2004*. IEEE, 2004.
13. B. Liskov, R. Atkinson, T. Bloom, E. Moss, J.C.Schaffert, R. Scheiffer, and A. Snyder. *CLU reference manual*. Springer Verlag, 1981.
14. B. Liskov, M.Day, M. Herlihy, P. Johnson, and G. Leavens. ARGUS reference manual. Technical report, MIT, 1987.
15. S. Murer, S. Omohundro, D. Stoutamire, and C. Szyperski. Iteration abstraction in Sather. *ACM Transactions on Programming Languages*, 18(1):1–15, 1996.
16. S. Murer, S. Omohundro, and C. Szyperski. Sather iters: object-oriented iteration abstraction. Technical Report TR-93-045, August 1993.

17. Spec# project web page. URL: http://research.microsoft.com/SpecSharp/.
18. Grant Richins. Recursive iterators (aka perf killers). URL: http://blogs.msdn.com/grantri/archive/2004/03/24/95787.aspx, March 2004.
19. Grant Richins. Recursive iterators made iterative. URL: http://blogs.msdn.com/grantri/archive/2004/04/08/110165.aspx, April 2004.
20. Mary Shaw, William A. Wulf, and Ralph L. London. Abstraction and verification in alphard: Defining and specifying iteration and generators. *Communications of the ACM*, 20(8), 1977.
21. Jeanette Marie Wing. *A two-tiered approach to specifying programs.* PhD thesis, MIT, Lab. for Comp.Sci., 1983.

Appendix. Programming methodology soundness proof sketch

In this section, we provide a sketch of a proof of the soundness of our proposed programming methodology for avoiding interference between parties in an enumeration. The sketch focuses on the distinctive features of the proposed methodology compared with the Boogie methodology and its soundness proof given in [2].

We use the following auxiliary definitions:

$$(o \text{ owns } p) \stackrel{\text{def}}{=} (o.inv \wedge p \in \text{rep}(o)) \qquad X^* \stackrel{\text{def}}{=} \{p \mid o \in X \wedge o \text{ owns}^* \ p\}$$

Definition 1 (Operation). *An* operation *is an object creation, a field read, a field write, a pack, an unpack, a read block entry, a read block exit, a parallel execution start, or a parallel execution termination.*

Definition 2 (Program Execution). *A* program execution *is a finite sequence of operations $a_1, \ldots, a_n$. We distinguish* program execution points $0, 1, \ldots, n$. *If a thread t is alive at a program execution point i, then we distinguish the* thread execution point (t, i).

Definition 3 (Happens before). *The* happens before *relation is the smallest transitive relation that satisfies the following:*

- *A thread execution point (t, i) happens before each operation a_j performed by thread t where $i < j$.*
- *An operation a_i performed by thread t happens before each thread execution point (t, j) where $i \leq j$.*
- *A parallel execution start operation a_i that creates a thread t happens before (t, i).*

Two operations or execution points that are not ordered by the happens before relation are said to be *concurrent*.

Lemma 1 (Structural Properties). *Consider an execution by a thread t of a command, that extends between program points i and j. We have the following:*

- *$t.R_i = t.R_j$. That is, thread t's read bag is the same in the pre-state and in the post-state. In other words, no command has a net effect on the read bag.*
- *If (t, i) happens before (t', j) for some t', then $t = t'$. In other words, no threads that were started by the command execution survive after the post-state.*

Proof. By induction on the number of operations in the command execution and case analysis on the command.

Lemma 2 (Program Invariants). *At each program execution point i, we have the following:*

1. *If $t_1 \neq t_2$, then $t_1.R'^*$ and $t_2.W'^*$ are disjoint.*
2. *If o is in $t.R'^*$, then each preceding write of any field of o happens before (t, i). In other words, a thread's observable state is never influenced by concurrent threads.*
3. *If $t.W'[o]$, then for each preceding program execution point j where $t'.W'[o]$, it holds that (t', j) happens before (t, i). In other words, two thread execution points that have write access to an object are never concurrent.*
4. *If $t.R[o] > 0$, then for each preceding read block entry operation a_j on o that brings $t'.R[o]$ from 0 to 1, for some t', a_j happens before (t, i). In other words, if at some point a thread has shared read access to an object, then this thread execution point is not concurrent with any preceding operation that causes the object to become shared.*

Proof. By induction on the length of the execution, and case analysis on the last operation performed. The interesting case is the read block exit operation. Suppose it brings $t.R[o]$ to zero. It follows that at the program point j after the operation, we have $t.W'[o]$. We prove by contradiction that it does not conflict with any other thread's effective read set. Suppose that $t'.R'[o]$ and $t \neq t'$. It follows that $t'.R[o] > 0$. By the fourth Program Invariant applied to the execution point $j - 1$ preceding the read block exit operation, we know that the corresponding read block entry happens before $(t', j - 1)$. Therefore, the start of the execution of the body of the read block also happens before $(t', j - 1)$. We now have a contradiction with the second Structural Property, which says that t' cannot be alive.

Theorem 1 (Soundness). *Whenever a thread t performs a read operation on a field $o.f$, the most recent write to $o.f$, if any, happens before the read. In other words, no thread ever sees writes performed concurrently by other threads.*

Proof. Follows from the second Program Invariant.

Corollary 1. *It is sound to verify a* for-each *loop and the enumerator method it calls as if they executed separately.*

Proof. Since neither party sees writes performed by the other, we may assume that none occur.