Mono's System.Collection Classes: A Spec# Case Study

Benjamin Lutz

Semester Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

http://sct.inf.ethz.ch/

September 2006

Supervised by:
Arsenii Rudich
Prof. Dr. Peter Müller

Software Component Technology Group
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Spec# is a programming system by Microsoft Research that extends the C# programming language with specification and verification capabilities. An important part of Spec# is Boogie, a tool for static verification of a given program's specification. With Boogie it is possible to automatically prove the correctness of a program.

This case study is about applying Spec# to code from Mono's System.Collections classes, the emphasis being on learning about today's specification and verification possibilities as well as determining how well Spec# works with real life code.

As part of this case study, a tutorial for turning C# programs into Spec# ones is developed.

Contents

1. Introduction

A big problem in software construction these days is that of correctness. The programs we have today are enormously complex, ranging into millions of lines of code, yet the methods we use today for ensuring the correctness of that software seem weak and arcane. There are "safe programming guidelines", libraries that present safe interfaces and often code is thoroughly tested, yet it can be observed every day by computer users everywhere that these methods are woefully inadequate: programs keep crashing and are plagued by security issues.

Static verification offers a better approach: the programmer, instead of being diligent and testing his code until he's reasonably confident that it's error free, can prove his code to be correct. He does so by augmenting his program with specifications and running it through a static verifier which attempts to prove that the program, in all conditions, obeys that specification.

One such specification and verification system is the *Spec# programming system* by Microsoft Research [1]. Spec# is an extension of the popular C# programming language, so it is possible to enhance a given C# program with specifications with little effort, turning it into a Spec# program which can then be verified. Spec# also comes with *Boogie*, a verifier that allows proving the correctness of Spec# programs.

As a case study of how well the Spec# programming system can be applied to existing, real life C# code, I have selected three classes from the Mono Project's [2] System.Collections framework, and after turning them into Spec# code, tried to prove their correctness. Mono is an open source implementation of Microsoft's .NET that has been selected for this case study because the source code is freely available. The System.Collections framework contains container classes and interfaces like lists, stacks and queues. It is part of the Mono and Microsoft .NET core libraries and thus is widely used in many different programs.

The goals of this case study are:

1. Learning about current specification and verification technology. Spec# is an active research project which includes and develops many novel and modern ideas that go beyond what was previously thought of when talking about adding specifications to programs. It is interesting to see what we can do today, and where Spec# still falls short.
2. Verifying that the analyzed Mono code is correct, or finding and fixing errors. Being part of the Mono/.NET core libraries, the System.Collections classes are very well tested. I expect no obvious errors and mistakes, however until there is proof, we cannot be sure that the code is totally correct. Using Spec# I will either prove that the code is correct, or find bugs that have remained despite intensive testing.
3. Testing Spec# and finding and reporting bugs. Spec# is still in heavy development, therefore many issues have to be expected. By analyzing and reporting them, I hope to help improve it.
4. Developing a systematic methodology for finding and applying specifications to existing C# code. The intention is to find a set of guidelines that make the process of finding Spec# specifications for a given program easier and faster. The results will be presented in tutorial form that is hopefully useful for other Spec# programmers.

2. An Overview over Spec#

2.1. The Spec# Programming Language

Spec# is a C# extension that introduces several new concepts and language features. This is a non-exhaustive overview over the most important ones. More details are available in [3, 4, 5].

Non-Null Types

A simple yet effective feature for improving program correctness are non-null types. Reference types may be marked as non-null, which prevents **`null`** from being assigned to objects of such a type, and therefore, null-dereference errors. This is an extension of the type system: Non-null types are a static type, and their type-safety is enforced by the compiler. Non-null types are "backwards compatible": a non-null type object may be assigned to a possibly-null type object, but the reverse is prohibited.

A type is marked as non-null by appending a `!` character to it. See listing 2.1 for an example.

Method Contracts

Putting method contracts in the actual program code is an old idea that has been most prominently implemented by Bertrand Meyer in his programming language Eiffel, under the name *Design by Contract* [6]. Class methods are provided with pre- and postconditions, which check for program errors. Preconditions make sure that the context a method runs in is valid. It is the responsibility of the caller to establish the precondition, and thus any precondition violation points to an error in the calling code.

Postconditions are guaranteed by the method to hold at method exit. A postcondition violation thus points to an error in the method.

In Spec#, preconditions are given using the **`requires`** keyword, while postconditions are given with the **`ensures`** keyword. Optionally, a custom exception may be specified to be thrown if a precondition is violated using the **`otherwise`** keyword. In postconditions, one may refer to the return value with the **`result`** keyword, and to the values that parameters had at the beginning of the method with the **`old`** keyword. See listing 2.2 for an example.

```
object a; // standard possibly-null type
object! b; // non-null type

a = b; // ok
b = a; // type error, doesn't compile
```

Listing 2.1: Non-null Types

```
public int modulo(int dividend, int divisor)
   requires divisor != 0 otherwise ArgumentNullException; // precondition
   ensures result < divisor; // postcondition
   ensures dividend == old(dividend); // bogus postcondition, just shows the old keyword
{
   return dividend % divisor;
}
```

Listing 2.2: Method Contracts

The Spec# compiler will insert code that checks pre- and postconditions at runtime, however it is also possible to use them for static verification with Boogie.

An important aspect of pre- and postconditions is that they are inherited from base classes and interfaces. Also, at the time of this writing, a child class may not modify the preconditions for an inherited method (even those that it overrides). This implementation is stricter than other implementations which allow overriding methods to weaken the precondition.

There are restrictions in what pre- and postconditions may refer to. Only pure methods may be called in them, i.e., methods that don't modify the object state[1]. Also, since they are part of the interface of a method, pre- and postconditions may not refer to methods or fields with a higher access level than their containing method: pre- and postconditions in a **public** method may not refer to methods or fields marked as **protected** or **private**.

Invariants

Invariants are similar to postconditions in that they ensure the correctness of the code that equips them. Invariants are used in class context and specify conditions that must hold throughout the lifetime of an instance of that class. They are not part of a class' public interface, and therefore may refer to private fields and methods of a class.

Spec# adds a special, boolean field, `inv`, to classes which specifies whether the invariant currently holds. If the invariants of an object holds, it is said to be *consistent*. While `inv` is **true**, the fields of an object may not be modified, since that could break an invariant. If an update of those fields is necessary, Spec# requires the programmer to **expose** the object. At the beginning of an **expose** block, `inv` is set to **false**. At the end of the **expose** block, the invariant is checked. If it holds, `inv` is reset to **true**, otherwise an exception is thrown. It is not possible to return from a method while **this** is exposed, nor can methods be called that require **this** to be consistent. This system ensures that outside of methods that modify it, an object is always in a consistent state, i.e., its invariants hold. For more details on **expose**, see the description of the `[Additive]` attribute below.

A second type of invariant is the loop invariant. It specifies conditions that must hold during the execution of a loop. A small but important detail is that loop invariants are validated before the loop condition is checked. It is therefore not possible to specify the loop condition as loop invariant, since after the last iteration of the loop, when the loop condition is false, the invariant is still required to hold.

Both types of invariants are specified with the **invariant** keyword. See listing 2.3 for an example that shows the use of invariants. See listing 2.4 for an **expose** example.

[1] The reason for this is that pre- and postconditions may be omitted by the compiler – the program should not change its behaviour in this case. The same applies to **assert** statements, which may or may not be executed.

```
public class Example {
   private int a;
   private int b[];

   invariant a < 0; // a is always negative

   public Example() {
      a = -1; // constructor must establish invariant

      b = new int[10];
      for (int i = 0; i < 10; i++)
         invariant i > 0 && i <= 10; // different from loop condition
      {
         b[i] = i + 1;
      }
   }
}
```

Listing 2.3: Invariant

```
public class Example {
   private int a;
   private int b;

   invariant a == b;

   // increment both a and b
   private increment() {
      // incrementing a and b sequentially would break the invariant.
      // Spec# offers the expose mechanism to handle this.
      expose (this) {
         a++;
         b++;
      }
   }
}
```

Listing 2.4: **expose** Blocks

Object Ownership

Spec# has the concept of object ownership. The idea is that every object has at most one owner, and only the owner may effect modifications of that object. References are therefore augmented with a type classification that expresses the ownership relationship between the two involved objects. There are three types of references:

rep Rep (for representation) references express that the object holding the reference is the owner of the referenced object. An owner is free to do anything it wants with an object it owns. It may also transfer the ownership. In Spec#, rep is the default reference type for class fields.

read-only An object may also hold references to objects that it doesn't itself own. These references are readonly references; they restrict method calls to pure methods and prevent field assignments. Read-only references are marked with `[Owned(false)]`.

peer If a group of objects have the same owner, they are considered peers. Peer objects may reference each other with peer references, and they are allowed to modify each other. Peer references are marked with `[Owned("peer")]`.

An object may be unowned, which means that all references to it are read-only references. An unowned object has no peers.

Spec# uses the concept of *peer consistency*. An object is peer consistent, if it is consistent, and all of its peer objects are consistent.

Assertions and Assumptions

Spec# adds support for assertions with the **`assert`** keyword. Assertions are boolean expressions that are meant to evaluate to **`true`**. The Spec# compiler optionally generates code for assertions that checks their expression at runtime. Since evaluating assertions is optional, the same rule applies to them that applies to method contracts: they must not modify anything.

Assumptions are instructions meant for Boogie (see section 2.2). They express "facts" that Boogie will then assume as valid. Once Boogie is complete, assumptions will probably not be used much anymore, but in the meantime it is sometimes necessary to give Boogie certain hints about the code it's trying to prove. Assumptions use the **`assume`** keyword, and as with assertions, they may not modify anything.

Assumptions have to be specified carefully, since they can prevent Boogie from correctly proving the code. It is for example possible to specify **`assume false`**, after which anything is correct, as far as Boogie is concerned.

See listing 2.5 for an example that demonstrates the use of **`assert`** and **`assume`**. The `==>` operator the example uses is the implication operator. `p ==> q` is equivalent to `p || !q`.

Attributes

Spec# introduces a number of attributes. I will present an overview over the ones of consequence to this case study. There are other attributes that I don't describe here, some of which are used only internally by the Spec# compiler and Boogie.

```
public int modulo(int dividend, int divisor)
   requires divisor != 0 otherwise ArgumentNullException; // precondition
   ensures result < divisor; // postcondition
{
   int result;
   assert divisor != 0; // redundant assertion
   result = dividend % divisor;
   // this assumption may or may not actually be true — Boogie
   // will accept it as true either way.
   assume dividend >= 0 ==> result >= 0 && result < divisor;
}
```

Listing 2.5: Assertions and Assumptions

[Additive] There are two ways to expose an object, additive and non-additive exposes. An object is divided into *class frames*. A class frame contains all fields and methods defined in a class, and the class frame of the class' base class. **expose** only exposes a single class frame. For a class frame to be exposable for an additive expose, its containing class frame needs to be exposed. The syntax for this type of expose is **expose** `(object) { ... }`. Non-additive exposes do not have this requirement, they may expose any class frame, as long as the object being exposed is peer consistent. The syntax for a non-additive expose is: **expose** `(object` **at** `classname) { ... }`. To express which sort of **expose** will be used in them, methods may be marked with `[Additive(`**true**`)]` or `[Additive(`**false**`)]`. The default currently is `[Additive(`**true**`)]`, however the Microsoft Spec# team plans to change the default for virtual methods to `[Additive(`**false**`)]`. For a more detailed explanation of additive versus non-additive exposes, see [7].

[Captured] By default, objects passed as method parameters are passed as read-only references. When an ownership change is necessary, for example to make **this** a peer of the object passed as parameter, the method needs to be marked with `[Captured]`.

[NotDelayed] By default, non-null field initialisation in Spec# is delayed, so constructors may not read from non-null fields, but only assign to them. If it is nevertheless necessary to read from non-null fields, the constructor must be marked as `[NotDelayed]`, and the fields being read need to be statically initialized.

[Owned] The three ownership modes described earlier are expressed in the Spec# code with `[Owned(`**true**`)]` for rep references, `[Owned("peer")]` for peer objects, and `[Owned(`**false**`)]` for read-only objects. `[Owned(`**true**`)]` is the default and may therefore be omitted.

[Pure] Pure methods are methods that have no observable side effects. Methods that are referred to in assertions, pre- and postconditions must be pure, since these constructs may or may not be executed at runtime.

[SpecPublic] For the purposes of verifying a class, it is possible to modify the access level of fields. A private field marked as `[SpecPublic]` may be referred to in method contracts of public methods. At the moment this is mostly used to work around deficiencies in Spec#. The Spec# team says that `[SpecPublic]` might also be used when a field has been marked as private to prevent a client from updating it, but the client should still know about the field. Since C# features accessors, this seems unnecessary.

[Verify] Boogie (see section 2.2) can take a long time to verify a given class. When methods of that class are marked as `[Verify(`**false**`)]`, Boogie will skip them, thereby reducing the time needed for the verification of the class. I've used this during development to skip

methods that have already been proved correct. The default for all methods is of course `[Verify(true)]`.

2.2. Boogie

Boogie is one of the most interesting aspects of the Spec# programming system. It is a static verifier that attempts to statically prove a program to be correct by translating a given Spec# program into first order logic code, *BoogiePL* (BPL), and running a theorem prover. For the program to pass this static validation, the specification of every method of the program has to be observed wherever the method is called, and with all possible arguments.

Boogie's analysis is complete. This means that wherever a certain value is unknown because it's determined at runtime, Boogie considers all possible values. A program verified in this way will always observe its specifications when running, barring any abnormal conditions. With this it is possible to detect program errors that might otherwise go unnoticed for a long time, since they might be very hard to come across in testing. Take for example a routine that divides an number by a random 32 bit integer. If the programmer has forgotten the check for zero, this error might slip through testing because the probability of it manifesting itself is very low. Boogie however will do an analysis of the possible range of values of that integer number, find that the value could possibly be zero at the division, and therefore give an error message.

Boogie takes compiled code that is equipped with debugging information as input. The BoogiePL program it creates can optionally be output, since it might give the programmer additional insight. At the moment, the theorem prover that Boogie employs is *Simplify*, the theorem prover from ESC/Java [8], however the Microsoft Research team is currently developing their own prover to replace it.

3. Reviewed Code

3.1. Conventions Used in this Chapter

Most of the code that I'll show in this section comes from Mono's System.Collection classes. Since one of my goals was not to change the original code if possible, and to only add specifications, I've used the following convention in the code.

All C#-compatible code is code coming directly from Mono's source distribution, version 1.1.15 (released in April 2006); this includes all comments. Spec# constructs like **`assume`**, **`ensures`**, **`invariant`**, **`requires`**, etc., were added by me. Comments starting with *//!* were also added by me. If it was necessary to change the original Mono code, this is noted in *//!* comments.

This chapter will present many excerpts from the code. To find the full, uninterrupted files, see appendix A. The line numbers in both places correspond.

A note: two new Spec# versions were released while I was working on this case study. Each brought new features and enhancements, but unfortunately also changed the behaviour of the compiler and Boogie slightly. Therefore, future Spec# versions might not compile this code cleanly. The last Spec# version that I used and verified the code with is 1.0.6404, the corresponding Boogie version is 0.80.

3.2. BitArray

Fields and Invariant

BitArray is a class that stores arrays of bits. It uses an integer array, `m_array`, for storage and therefore packs 32 bits into each element of the array. There is an integer variable, `m_length`, which tracks the length of the bit array, that is, the number of bits currently stored. See listing 3.1 for the definition of these fields.

```
42    public sealed class BitArray : ICollection, ICloneable {
43        //! initialisation necessary because of not-delayed constructors.
44        //! old code was: int [] m_array;
45        int []! m_array = new int[0];
46        [SpecPublic] int m_length;
47        int _version = 0;
48
49        invariant m_length >= 0;
50        invariant m_array.Length >= (m_length + 31) / 32;
51        invariant m_array != null;
```

Listing 3.1: BitArray Fields and Invariant

```
68      [NotDelayed]
69      public BitArray (bool []! values)
70      {
71          //! if (values == null)
72          //!     throw new ArgumentNullException ("values");
73
74          m_length = values.Length;
75          m_array = new int [(m_length + 31) / 32];
76          assume m_array.Length == (m_length + 31) / 32;
77
78          //! Added temp vars to express that m_length and m_array.Length remain
79          //! constant through the for loop.
80          int temp1 = m_length;
81          int temp2 = m_array.Length;
82
83          for (int i = 0; i < values.Length; i++)
84              invariant temp1 == m_length;
85              invariant temp2 == m_array.Length;
86          {
87              this [i] = values [i];
88          }
89      }
```

Listing 3.2: Constructor

Based on this, I defined the invariant: `m_length` must be non-negative, and the storage array `m_array` must always have enough room for `m_length` bits. This links `m_array` and `m_length` together so that it is always ensured that one variable's value makes sense given the other's value. The third clause, saying that `m_array` cannot be **`null`** is redundant, since `m_array` was declared as a non-null type, however, without it, Boogie claims that `m_array.Length`, used inside for loop conditions, may cause a null-dereference. This is a fault in Boogie, and will be corrected in a future version.

Please note one change for `m_array` that I've introduced: I have added an initialisation. Since there are not-delayed constructors, it is required that `m_array` be initialized at the beginning of the constructors. Because there is no default constructor for non-null array references, I have to explicitly specify one.

Another thing to note is that `m_length` has been marked `[SpecPublic]`. A deficiency of the current Spec# tools prevents methods from being used in method contracts. Since I will need to refer to the length of the bit array in public method contracts later on, using `m_length` works too, and the `[SpecPublic]` attribute relaxes the permissions enough so that Spec# lets me do that. In the future, the correct way to do this would be to use the `Count` member instead.

Constructors

BitArray comes with a number of constructors that are very similar. I will therefore just show three representative ones here.

This first constructor (listing 3.2) takes values from a boolean array. The check for **`null`**-ness of `values` is unnecessary now, since I made its type non-null. Calling this constructor with a standard, possibly-**`null`** array would not even compile, therefore this runtime check has been made redundant by the Spec# system.

```
126     [NotDelayed]
127     public BitArray (int length)
128        requires length >= 0 otherwise ArgumentOutOfRangeException;
129     {
130        //! if (length < 0)
131        //!      throw new ArgumentOutOfRangeException ("length");
132
133        m_length = length;
134        m_array = new int [(m_length + 31) / 32];
135     }
```

Listing 3.3: Constructor

```
137     [NotDelayed]
138     public BitArray (int length, bool defaultValue) : this (length)
139        requires length >= 0;
140     {
141        if (defaultValue) {
142           for (int i = 0; i < m_array.Length; i++)
143           m_array[i] = ~0;
144        }
145     }
```

Listing 3.4: Constructor

I also added two temporary variables to be used in the loop invariant. Boogie does not infer as much information into loops at the moment as one would hope, therefore I use this technique to help it along a bit. What the temporary variables do is express that the array lengths remain constant. Since I'm calling methods on both `this` (with `this [i] = values[i]`) and `values` (with `values.Length`), Boogie makes no assumptions about some fields of the objects not changing. By defining this loop invariant however, I establish that information inside the loop. Note that this invariant is not an assumption: Boogie does check whether it holds.

In listing 3.3 you see a classical example of how an implicit precondition in the form of a runtime check can be made explicit. Instead of checking whether `length` is negative, I can require that it is non-negative. The compiler will generate code that checks this condition at runtime. There are two advantages to using proper Spec# preconditions here: First, Given suitable documentation-generating tools, this precondition becomes part of the method documentation in an automatic way that prevents code/documentation desynchronization. Second, with Boogie's help can now prove that a given program always observes this precondition, without actually having to test it.

In the last constructor (listing 3.4), I specify the same precondition, but the C# code didn't explicitly check for it. In this case, the precondition comes from the initializer `this (length)`. Since I call the `BitArray(int length)` constructor, I must comply with its precondition. `length` is a parameter of this constructor, so the right way to do this is to repeat the precondition here.

For three of the constructors, `BitArray(bool[]! values)`, `BitArray(byte[]! bytes)` (not shown here) and `BitArray(int, bool)`, Boogie currently returns the an error message saying that the constructor should leave the receiver object in an unshared state. I have not been able to determine the cause for this message, or how to suppress it. While I believe that this error message is bogus and can be disregarded, there is one unfortunate consequence: after an error, Boogie stops checking the function that the error appeared in. It is therefore possible that these errors mask further errors.

```
157       [Pure]
158       byte getByte (int byteIndex)
159          requires byteIndex >= 0 && byteIndex < (m_length + 7) / 8;
160       {
161          int index = byteIndex / 4;
162          int shift = (byteIndex % 4) * 8;
163
164          int theByte = m_array [index] & (0xff << shift);
165
166          return (byte)((theByte >> shift) & 0xff);
167       }
168
169       void setByte (int byteIndex, byte value)
170          requires byteIndex >= 0 && byteIndex < m_length / 8;
171       {
172          int index = byteIndex / 4;
173          int shift = (byteIndex % 4) * 8;
174
175          // clear the byte
176          m_array [index] &= ~(0xff << shift);
177          // or in the new byte
178          m_array [index] |= value << shift;
179
180          _version++;
181       }
```

Listing 3.5: getByte and setByte Methods

getByte and setByte Methods

In the C# code, the `getByte` and `setByte` (see listing 3.5) methods don't contain checks for the validity of the `byteIndex` parameter, since the two methods only used internally and the author of the class appears to have sufficient confidence in his programming skills. I have added preconditions that constrain the range of `byteIndex` to the indices of used bytes in of `m_array`. For proving that `getByte` and `setByte` calls never cause an `IndexOutOfRange` Exception, this is not actually necessary since Boogie sees how `byteIndex` is used and infers its range constraints itself. However `m_array` may be bigger than the bit array stored in it, so the `byteIndex` constraints need to be stricter.

Boogie verifies that all `getByte` and `setByte` calls indeed always use valid parameters, so the class author's confidence is justified.

The difference in the upper limit for `byteIndex` between `getByte` and `setByte` exists because the bit array may contain a number of bits not divisible by 8. In that case, the last byte will only be partially filled with valid bits. When reading bits with `getByte`, this is OK: the returned byte will have some valid bits and some with an undefined state. When writing bits however, there would be "surplus bits" that would likely be overwritten at a later point. Since this means a loss of information in the case where the programmer is unaware that some of his bits might be discarded, my precondition does not allow that. This is my decision, the original code does not specifically handle surplus bits: they would simply be written to the storage array, past the logical size of the bit array. Using the same precondition in `setByte` as in `getByte` would therefore work too, technically. However, I prefer the semantically safer precondition, and Boogie proves that it

```
183         [Pure]
184         void checkOperand (BitArray! operand)
185             requires operand.m_length == m_length otherwise ArgumentException;
186         {
187             //! not necessary because operand is non-nullable.
188             //! if (operand == null)
189             //!     throw new ArgumentNullException ();
190
191             //! if (operand.m_length != m_length)
192             //!     throw new ArgumentException ();
193         }
```

Listing 3.6: checkOperand Method

is satisfied everywhere; the Mono code does not assume or require that writing to partially-used bytes is allowed.

checkOperand Method

The `checkOperand` method, presented in listing 3.6, beautifully demonstrates the ideas behind Spec#. In the C# version of BitArray, it was deemed necessary to write a dedicated function to check the validity of a value. These checks can be removed in Spec#, instead I use a non-null type and specify a precondition. The result is less explicit error checking cluttering the code (without giving up any robustness), implicit documentation, and of course, verifiability.

The method could easily be removed altogether: when `checkOperand` is called, it is always at the beginning of a function and with a parameter of that function. To call `checkOperand` and observe its preconditions, those functions therefore require at least the same preconditions as `checkOperand` itself, so the actual call of `checkOperand` is redundant, since `checkOperand` is now empty.

Indexing Operator

Since the indexing operator (listing 3.7) only forwards the `get` and `set` calls to the `Get` and `Set` methods (see listing 3.10), and it must observe the preconditions of those two functions, these preconditions apply to it as well. They are therefore simply repeated here.

The getter can be marked as `[Pure]` because the `Get` method is pure too, and no other (possibly non-`[Pure]`) method is called.

Length Accessors

In the `Length` setter (listing 3.8) we have the first example of an **`expose`** block. As explained in section 2.1, an object needs to be explicitly exposed for write access. The rules Spec# uses to determine where **`expose`** blocks are necessary are somewhat unclear. In some cases, Boogie complains when the **`expose`** block is missing, other times it's fine. The default contract for public methods in Spec# exposes objects by default, however here Boogie complains if it's missing. It seems that **`this`** is not exposed with setters.

```
212      public bool this [int index] {
213         [Pure]
214         get
215            requires index >= 0 && index < m_length otherwise
                   ArgumentOutOfRangeException;
216            requires IsPeerConsistent;
217         {
218            return Get (index);
219         }
220         set
221            requires index >= 0 && index < m_length otherwise
                   ArgumentOutOfRangeException;
222         {
223            Set (index, value);
224         }
225      }
```

Listing 3.7: Indexing Operator

```
227      public int Length {
228         [Pure]
229         get { return m_length; }
230         set
231            requires value >= 0 otherwise ArgumentOutOfRangeException;
232         {
233            //! if (value < 0)
234            //!      throw new ArgumentOutOfRangeException ();
235
236            int newLen = value;
237            if (m_length != newLen) {
238               int numints = (newLen + 31) / 32;
239               int [] newArr = new int [numints];
240               int copylen = (numints > m_array.Length) ? m_array.Length : numints;
241               expose (this) {
242                  assume m_array.IsPeerConsistent;
243                  Array.Copy (m_array, newArr, copylen);
244
245                  // set the internal state
246                  m_array = newArr;
247                  m_length = newLen;
248                  _version++;
249               }
250            }
251         }
252      }
```

Listing 3.8: Length Accessors

There also in an **assume** `m_array.IsPeerConsistent` inside the **expose** block. `m_array` being peer consistent is a precondition of `Array.Copy`. Boogie isn't able to prove this though, so I've provided this **assume** statement so that Boogie can proceed with checking the method.

CopyTo Method

The `CopyTo` Method (see listing 3.9) looks fairly complex, since it consists of four different functions rolled into one. Again, a lot of error checking code can be removed in favor of preconditions.

There are a number of invariants and assumptions inside **for** loops again, which are required because Boogie, at the moment, is poor at inferring properties of variables inside loops. For example, the loop invariant in line 306 expresses that `numbytes` remains constant. This is obvious since there is no write access to it, however Boogie doesn't figure that out yet.

Having to use an **assume** in line 308 is unfortunate, since in contrast to loop invariants, assumptions are not checked, but accepted in "blind faith". As explained in section 2.1, the reason for this is that loop invariants must hold even after the last iteration of the loop, when the loop condition no longer is true. In other words, a loop invariant of `i < numbytes` would fail, since after the last loop iteration `i` is equal to `numbytes`. A loop invariant of `i <= numbytes` would fail too, because it would allow out-of-bounds access to `barray`.

The **else** block at line 314 can be omitted altogether because it follows from the precondition in line 269 that if execution of the method body begins (the precondition was observed), one of the **if** blocks will be chosen.

Boogie returns one error message about `CopyTo` whose meaning is unclear:

> Call of System.Collections.BitArray.CopyTo(System.Array! array, int index), unsatisfied precondition: requires index < array.Length;

Since `CopyTo` is never called inside the BitArray class, I'm suspecting that this error message is bogus.

Get and Set Methods

Boogie is unaware of the mathematics of bitshifts, so it was necessary to add assumptions to the Get and Set methods (see listing 3.10) which explain their behaviour. With these, the code validates. I've tried to find the weakest possible preconditions so as not to hide any possibly incorrect code from the verifier. I could also have said, as a somewhat extreme example, **assume false**; Boogie would then have gladly accepted the code – as it would have any other code, no matter how wrong it would have been.

```
265      public void CopyTo (Array! array, int index)
266         requires index >= 0 otherwise ArgumentOutOfRangeException;
267         requires array.Rank == 1 otherwise ArgumentException;
268         requires index < array.Length otherwise ArgumentException;
269         requires array is bool[]! || array is byte[]! || array is int[]!;
270         requires array is bool[]! ==> array.Length - index >= m_length;
271         requires array is byte[]! ==> array.Length - index >= (m_length + 7) / 8;
272         requires array is int[]! ==> index + (m_length + 31) / 32 <= array.Length;
273      {
274         //! if (array == null)
275         //!     throw new ArgumentNullException ("array");
276         //! if (index < 0)
277         //!     throw new ArgumentOutOfRangeException ("index");
278         //! if (array.Rank != 1)
279         //!     throw new ArgumentException ("array", "Array rank must be 1");
280         //! if (index >= array.Length)
281         //!     throw new ArgumentException ("index", "index is greater than array.Length");
282
283         // in each case, check to make sure enough space in array
284         if (array is bool []!) {
285            //! if (array.Length - index < m_length)
286            //!     throw new ArgumentException ();
287
288            bool []! barray = (bool []!) array;
289
290            // Copy the bits into the array
291            for (int i = 0; i < m_length; i++)
292               invariant array.Length - index >= m_length; //! precondition
293            {
294               assume i < m_length; //! loop stop condition
295               barray[index + i] = this [i];
296            }
297         } else if (array is byte []!) {
298            int numbytes = (m_length + 7) / 8;
299
300            //! if ((array.Length - index) < numbytes)
301            //!     throw new ArgumentException ();
302
303            byte []! barray = (byte []!) array;
304            // Copy the bytes into the array
305            for (int i = 0; i < numbytes; i++)
306               invariant numbytes == (m_length + 7) / 8;
307            {
308               assume i < numbytes; //! loop stop condition
309               barray [index + i] = getByte (i);
310            }
311         } else if (array is int []!) {
312            assume m_array.IsPeerConsistent;
313            Array.Copy (m_array, 0, array, index, (m_length + 31) / 32);
314         //! } else {
315         //!     throw new ArgumentException ("array", "Unsupported type");
316         }
317      }
```

Listing 3.9: CopyTo Method

```
384       [Pure]
385       public bool Get (int index)
386          requires index >= 0 && index < m_length otherwise
                 ArgumentOutOfRangeException;
387       {
388          //! if (index < 0 || index >= m_length)
389          //!     throw new ArgumentOutOfRangeException ();
390
391          assume index >= 0 ==> (index >> 5) >= 0;
392          assume (index >> 5) == (index / 32);
393          return (m_array [index >> 5] & (1 << (index & 31))) != 0;
394       }
395
396       public void Set (int index, bool value)
397          requires index >= 0 && index < m_length otherwise
                 ArgumentOutOfRangeException;
398       {
399          //! if (index < 0 || index >= m_length)
400          //!     throw new ArgumentOutOfRangeException ();
401
402          assume index >= 0 ==> (index >> 5) >= 0;
403          assume (index >> 5) == (index / 32);
404          if (value)
405             m_array [index >> 5] |= (1 << (index & 31));
406          else
407             m_array [index >> 5] &= ~(1 << (index & 31));
408
409          _version++;
410       }
```

Listing 3.10: Get and Set Methods

```
44  public class Queue : ICollection, IEnumerable, ICloneable {
45
46      private object[]! _array;
47      private int _head = 0; // points to the first used slot
48      [SpecPublic] private int _size = 0;
49      private int _tail = 0;
50      private int _growFactor;
51      private int _version = 0;
52
53      invariant _head >= 0;
54      invariant _array.Length > 0 ==> _head < _array.Length;
55      invariant _array.Length == 0 ==> _head == 0;
56      invariant _tail >= 0;
57      invariant _array.Length > 0 ==> _tail < _array.Length;
58      invariant _array.Length == 0 ==> _tail == 0;
59      invariant _size >= 0 && _size <= _array.Length;
60      invariant _array.GetType() == typeof(object[]);
```

Listing 3.11: Queue Fields and Invariant

3.3. Queue

Fields and Invariant

The Queue class implements a queue of objects on top of an array, `_array` (see listing 3.11). There are two pointers, `_head` and `_tail`, that point to the elements on `_array` that are the head and tail of the queue. As elements are added and removed from the queue, these pointers continuously move forward on the array, wrapping around to its beginning if they reach its end. If the storage array is full, but additional elements are added to the queue, a new, larger array is allocated (how much larger is influenced by `_growFactor`), and the existing elements are copied to it. There is further a field `_size` which keeps track of the number of elements stored in the queue.

The invariant first defines the valid range of `_head` and `_tail`. They are always non-negative, but smaller than `_array`'s length only when there are elements in the queue. If the queue is empty, and thus there are no elements, `_head` and `_tail` don't really make sense; the Mono code just sets them to 0 in this case.

The last part of the invariant which refers to `_array.GetType` is there for Boogie's benefit. Arrays in C# (and thus in Spec#) are covariant, so a reference to an `object[]` array might really hold a `string[]` array, in which case assigning an `object` to it isn't allowed. Boogie at the moment does not infer the real type of `_array` from its initialisations in the constructors, so I state the real type in the invariant here.

Constructors

Most `Queue` constructors (see listing 3.12) take an `initialCapacity` argument, which must be non-negative. This is easily expressed as a precondition, making an explicit check unnecessary.

There are two problems with the constructors. First, as with BitArray, Boogie returns an error message saying that the constructor `Queue(ICollection! col)` should leave the receiver object in an unshared state. Also, Boogie currently does not handle floating point numbers. I've therefore replaced assignment to `_growFactor` (line 92) with a default, integer expression.

```
62      public Queue () : this (32, 2.0F) {}
63      public Queue (int initialCapacity) : this (initialCapacity, 2.0F)
64         requires initialCapacity >= 0;
65      {}
66      public Queue(ICollection! col) : this (col == null ? 32 : col.Count)
67         requires col.IsPeerConsistent;
68      {
69         //! if (col == null)
70         //!     throw new ArgumentNullException ("col");
71
72         // We have to do this because msft seems to call the
73         // enumerator rather than CopyTo. This affects classes
74         // like bitarray.
75         foreach (object o in col)
76            Enqueue (o);
77      }
78
79      public Queue (int initialCapacity, float growFactor)
80         requires initialCapacity >= 0 otherwise ArgumentOutOfRangeException;
81      {
82         //! if (initialCapacity < 0)
83         //!     throw new ArgumentOutOfRangeException("capacity", "Needs a non-negative
                number");
84
85         //! Can't run this code, since Boogie currently doesn't know anything about floats
86         //! if (!(growFactor >= 1.0F && growFactor <= 10.0F))
87         //!     throw new ArgumentOutOfRangeException("growFactor", "Queue growth factor
                must be between 1.0 and 10.0, inclusive");
88
89         _array = new object[initialCapacity];
90
91         //! Can't run this code, since Boogie currently doesn't know anything about floats
92         //! this._growFactor = (int)(growFactor * 100);
93         this._growFactor = 200;
94      }
```

Listing 3.12: Queue Constructors

```
122       [Additive(false)]
123       public virtual void CopyTo (Array! array, int index)
124       {
125          //! if (array == null)
126          //!      throw new ArgumentNullException ("array");
127
128          if (index < 0)
129             throw new ArgumentOutOfRangeException ("index");
130
131          if (array.Rank > 1
132             || (index != 0 && index >= array.Length)
133             || _size > array.Length - index)
134             throw new ArgumentException ();
135
136          int contents_length = _array.Length;
137          int length_from_head = contents_length - _head;
138          // copy the _array of the circular array
139          //! Boogie can't handle Math.Min(), replace with equivalent code
140          //! Array.Copy (_array, _head, array, index,
141          //!        Math.Min (_size, length_from_head));
142          int len = (_size < length_from_head ? _size : length_from_head);
143          assume _array.IsPeerConsistent;
144          Array.Copy (_array, _head, array, index, len);
145
146          if (_size > length_from_head)
147             Array.Copy (_array, 0, array,
148                   index + length_from_head,
149                   _size - length_from_head);
150       }
```

Listing 3.13: CopyTo Method

CopyTo Method

In the `CopyTo` method (see listing 3.13), there are again several explicit checks for preconditions in the C# code that look like they can be turned into Spec# preconditions. However, the `CopyTo` method is defined in the `ICollection` interface, so it's just being implemented here. This means that I can't add any additional preconditions, and thus the explicit checks have to stay.

Since Boogie currently doesn't know the properties of `Math.Min`, I have replaced it with an equivalent expression using the `(?:)` operator. This Boogie can handle.

GetEnumerator Method

The `GetEnumerator` method (see listing 3.14), although it is very short, unfortunately demonstrates two bugs in the Spec# compiler. First, the method is supposed to return an object that is a peer of **`this`**, and thus should get the `[Owned("peer")]` attribute. Unfortunately, the compiler currently gives the following bogus error message when specifying this attribute for methods:

> error CS2692: Methods without return value or with value-type return value must not be marked Owned.

```
154         //! currently prevented by compiler bug
155         //! [Owned("peer")]
156         public /*virtual*/ IEnumerator! GetEnumerator () //! virtual causes CS0029 error
157         {
158             return new QueueEnumerator (this);
159         }
```

Listing 3.14: GetEnumerator Method

However, `GetEnumerator` obviously does return a reference-type value. Removing the attribute makes the Spec# compiler compile the code, but then Boogie will give the following error message:

> unsatisfied postcondition: ensures Owner.Same(this, result)

At the moment, there is no way around this. The bug is confirmed and will be fixed in a future release of Spec#.

The second problem lies with the **`virtual`** keyword, which currently causes the following compiler warning:

> error CS0029: Cannot implicitly convert type 'System.Collections.Queue' to 'T'

As a consequence of removing the **`virtual`** keyword here, I've completely commented out the `GetEnumerator` method in the `SyncQueue` class (see below).

Clone Method

The clone method (see listing 3.15) returns a peer object, as shown by the `[Owned("peer")]` attribute. Unfortunately, due to the mentioned compiler bug, this doesn't work at the time of writing.

There are several **`assume`** statements before the `Array.Copy` call. Boogie does not infer the necessary properties itself, so I have to help it along. That `newQueue._array`'s length is indeed the same as the length of `_array` follows from the constructor, and is therefore a valid assumption.

Clear Method

The clear method (see listing 3.16) is interesting for its loop invariant. Boogie does not realize that `_head`, `_size` and `_tail` remain unchanged inside the **`for`** loop. Without the loop invariant that states this, it would complain that it can't prove the invariant at the end of the **`expose`** block.

The other part of the loop invariant expresses that `length` remains smaller than `_array`'s length, and therefore is a valid index. Again, Boogie requires me to explicitly state this a loop invariant. Interestingly, it does not require a statement about whether `length` is non-negative.

grow Method

The grow method (see listing 3.17) needs an `IsPeerConsistent` precondition, so that I can locally expose it. This precondition is not needed in the other methods shown here, since marking them as non-additive adds that precondition as a default.

Note also the postcondition, which uses the **`old`** keyword to refer to the value of `_array.Length` at method entry.

```
163                     //! currently prevented by compiler bug
164         [Additive(false)] //! [Owned("peer")]
165         public virtual object! Clone ()
166            ensures result is Queue;
167         {
168            Queue newQueue;
169
170            newQueue = new Queue (this._array.Length);
171            newQueue._growFactor = _growFactor;
172
173            //! This is established by the constructor
174            assume newQueue._array.Length == this._array.Length;
175            assume _array.IsPeerConsistent;
176            assume newQueue._array.IsPeerConsistent;
177            Array.Copy (this._array, 0, newQueue._array, 0,
178                  this._array.Length);
179            expose (newQueue) {
180               newQueue._head = this._head;
181               newQueue._size = this._size;
182               newQueue._tail = this._tail;
183            }
184
185            return newQueue;
186         }
```

Listing 3.15: Clone Method

```
188         [Additive(false)]
189         public virtual void Clear ()
190            ensures _size == 0;
191         {
192            expose (this at Queue) {
193               _version++;
194               _head = 0;
195               _size = 0;
196               _tail = 0;
197               for (int length = _array.Length - 1; length >= 0; length--)
198                  invariant length < _array.Length;
199                  invariant _head == _size && _head == _tail && _head == 0;
200               {
201                  _array [length] = null;
202               }
203               //! Boogie can't infer this in/after loop
204               assume _array.GetType() == typeof(object);
205            }
206         }
```

Listing 3.16: Clear Method

```
322      private void grow ()
323         requires IsPeerConsistent;
324         ensures _array.Length > old(_array.Length);
325      {
326         int newCapacity = (_array.Length * _growFactor) / 100;
327         if (newCapacity < _array.Length + 1)
328            newCapacity = _array.Length + 1;
329         object[] newContents = new object[newCapacity];
330         CopyTo (newContents, 0);
331         expose (this at Queue) {
332            _array = newContents;
333            _head = 0;
334            _tail = _head + _size;
335         }
336      }
```

Listing 3.17: grow Method

```
340      private class SyncQueue : Queue {
341         [Owned("peer")] Queue queue;
342         invariant queue != null;
343
344         [Captured] [NotDelayed]
345         internal SyncQueue (Queue! queue)
346            requires queue.IsPeerConsistent;
347            ensures IsPeerConsistent;
348         {
349            Owner.AssignSame(this, queue);
350            this.queue = queue;
351         }
```

Listing 3.18: Start of SyncQueue Class

SyncQueue Class

SyncQueue is a class that is private to Queue. It has a Queue object as a peer field, and forwards all method calls to this queue object after locking it. The reason for having this class is that it allows concurrent access to a single Queue without having to worrying about locking, as the SyncQueue class does that for the user. Unfortunately, this design doesn't work well with the current Spec# version for several reasons.

In the constructor (see listing 3.18), I need to "capture" the parameter to store in a peer reference field. I do this with the `Owner.AssignSame` call. Since this means referring to **`this`** in the constructor, I have to mark it with `[NotDelayed]`. You might be surprised to find the invariant `queue != null`, after all this could be expressed by using `Queue!` as the type for `queue`. However this would require me to initialize `queue`, which would then confuse Boogie in the `Owner.AssignSame` call - it's not clear which of the two object's owner should be the assigned as the other object's owner. (Implicitely) initializing it as null, and having an invariant to state the non-null-ness of `queue` works around this.

Furthermore, a number of **`assume`** statements are required for Boogie to be able to handle the code. `Queue`'s `Clear` method (see listing 3.19) has `queue._size == 0` as a postcondition. In the context of `SyncQueue`, this postcondition can't be fulfilled, however: SyncQueue doesn't actually

```
407         [Additive(false)]
408         public override void Clear () {
409            lock (queue) {
410               queue.Clear ();
411            }
412            assume queue._size == 0 ==> _size == 0;
413         }
```

Listing 3.19: SyncQueue.Clear Method

```
429         [Additive(false)]
430         public override object Dequeue () {
431            assume queue._size >= 1;
432            lock (queue) {
433               return queue.Dequeue ();
434            }
435         }
```

Listing 3.20: SyncQueue.Dequeue Method

have a _size field! If the postcondition were to refer to Count instead of _size, it would make sense in SyncQueue's context too, unfortunately, as mentioned before, this is not possible at the moment. I have therefore chosen to use a weak **assume** statement to help Boogie prove this code: if queue._size is zero after the Clear call (and it is, that's the postcondition of Clear), assume that SyncQueue._size is zero too.

I can refer to SyncQueue._size, even though it doesn't actually exist, because I've marked _size (in the Queue class) as [SpecPublic]. For the purposes of proving the code, _size is thus regarded as a public field, and SyncQueue inherits it.

SyncQueue.Dequeue is another example (see listing 3.20). Here we need an **assume** statement to establish Dequeue's precondition (please refer to appendix A.2 to see the code for Dequeue). This is needed because I've referred to _size in that precondition, not Count. Had I referred to Count, the same precondition would apply to SyncQueue.Queue as well, which would mean the right thing here. However, since Boogie doesn't evaluate method calls in method contracts, I can't refer to Count and still have it prove the code correct.

```
42  public class Stack : ICollection, IEnumerable, ICloneable {
43
44      // properties
45      private object[]! contents;
46      private int current = -1;
47      [SpecPublic] private int count;
48      private int capacity;
49      private int modCount;
50
51      const int default_capacity = 16;
52
53      invariant capacity == contents.Length;
54      invariant 0 <= count && count <= capacity;
55      invariant current == count - 1;
```

Listing 3.21: Stack Fields and Invariant

3.4. Stack

The `Stack` class is generally quite similar to the `Queue` class discussed in the previous section. I will therefore not go into it in as much detail, but only look at details that do not appear in `Queue`.

Fields and Invariant

The `Stack` class uses an `object` array, `contents` for storage. It comes with a counter `count`, and an index into the array, `current`. As expressed by the invariant, `current` is always one smaller than `count`. Further there is a `capacity`, which equals the length of the storage array. I've restricted `count` to numbers between 0 and `capacity`.

I've marked `count` as `[SpecPublic]` to be able to refer to it in method contracts. There is a public accessor for it, `Count`, but as discussed before I can't use it in method contracts due to deficiencies in Boogie.

Resize Method

The `Resize` method is used to enlarge the storage array. It creates a new array and copies the existing elements into it. Since stack elements should not be lost when resizing its underlying representation, I've specified the precondition that the user may not request the array to be resized to a length smaller than the number of elements currently stored. In turn the method will guarantee that the stack has a capacity of at least as many elements as requested (in some cases the capacity will be larger), and that the Stack functionally remains unchanged.

This is a case where I can give a full, functional contract for a non-trivial method, describing exactly what it does, and Boogie is actually able to prove that the code is correct. This is still a somewhat rare occurence, since often such contracts would require references to other methods (for example, the `Push` method of the stack should have the postcondition that the pushed object is the same as the one returned by `Peek`), which currently doesn't work. Let this method therefore be a preview of what Spec# will do in the future.

Note that Boogie doesn't know the properties of `Math.Max`. I've therefore replaced that line with equivalent code. With this change, Boogie can prove the method to be correct.

```
57      private void Resize(int ncapacity)
58         requires count <= ncapacity;
59         requires IsPeerConsistent;
60         ensures capacity >= ncapacity;
61         ensures count == old(count);
62         ensures forall { int i in (0:count - 1), contents[i] == old(contents[i])
               };
63      {
64         //! This was the original code here. However, it appears that
65         //! Math.Max() does not have any specifications, so I replaced
66         //! it with equivalent code that Boogie can handle.
67         //!ncapacity = Math.Max (ncapacity, 16);
68         if (ncapacity < 16) { ncapacity = 16; }
69
70         object[]! ncontents = new object[ncapacity];
71
72         assume contents.IsPeerConsistent;
73         Array.Copy(contents, ncontents, count);
74
75         expose (this at Stack) {
76            capacity = ncapacity;
77            contents = ncontents;
78         }
79      }
```

Listing 3.22: Resize Method

Enumerator Class

`Stack` provides an enumerator class (see listing 3.23) that enumerates the stack elements in a top-to-bottom manner. For this purpose it uses the field `current` as an index into the stack. When the enumerator is first used, `current` will be set to the topmost element index, and then decremented, until -1 is reached.

This behaviour unfortunately evades Boogie completely, requiring a curious amount of **`assume`**s in `Enumerator`'s `Current` method (see listing 3.24). It is unclear why Boogie fails here. For example, if I add the assertion

```
assert current >= -2 && current != -2 && current != -1;,
```

it will pass Boogie's check. However the assertion

```
assert current >= 0;,
```

which follows logically from the above assertion might not hold in Boogie's opinion.

```
360     private class Enumerator : IEnumerator, ICloneable {
361
362         const int EOF = -1;
363         const int BOF = -2;
364
365         [Owned("peer")] Stack stack;
366         private int modCount;
367         private int current;
368
369         invariant stack != null;
370         invariant current >= -2;
371
372         [Captured] [NotDelayed]
373         internal Enumerator(Stack! s)
374             ensures IsPeerConsistent;
375         {
376             Owner.AssignSame(this, s);
377             stack = s;
378             modCount = s.modCount;
379             current = BOF;
380         }
```

Listing 3.23: Start of Enumerator Class

```
387         public virtual object Current {
388             get {
389                 if (modCount != stack.modCount
390                     || current == BOF
391                     || current == EOF
392                     || current > stack.count)
393                     throw new InvalidOperationException();
394                 assert current >= 0;
395                 //! preceding if
396                 assume current >= 0 && current <= stack.count;
397                 //! current starts at stack.current, then is only decremented
398                 //! stack.current stays constant while modCount == stack.modCount
399                 assume current <= stack.current;
400                 assume stack.current == stack.count - 1; //! stack invariant
401                 assume stack.count <= stack.contents.Length; //! stack invariant
402                 assume stack.contents[current] != null ==> stack.contents[current].
                        IsPeerConsistent;
403                 return stack.contents[current];
404             }
```

Listing 3.24: Enumerator.Current accessor

4. Results

4.1. Bug in Mono's System.Collections.Queue Class

As expected, the Mono code was of high quality. With the help of Boogie, I did however find one bug in the System.Collections.Queue class. Under a certain condition, Queue objects throw an unexpected `IndexOutOfRangeException` because the code tries to write to a non-existant array element. Listing 4.1 shows a C# program that crashes with that exception.

Description

System.Collection.Queue stores its data in an array. It uses a head and a tail pointer (`_head` and `_tail`) to mark the array positions where elements are removed and added to the queue with the `Dequeue` and `Enqueue` methods. The head and tail pointer wrap around at the end of the storage array, so the array is effectively being used as cyclic storage area.

As the queue grows, larger storage arrays are allocated automatically, however the array size is never being automatically reduced. Instead, the `Queue` class offers the `TrimToSize` method which replaces the storage array with whose length equals the number of elements in the queue. When such an array reallocation happens, the queue elements are inserted into the array in order, ie., the oldest queue element (the queue head) is inserted at array position 0, and the newest queue element (the element one position below the queue tail) is inserted at the largest array position used. Since the position of queue elements on the storage array changes, the head and tail pointers need to be updated as well.

In the original TrimToSize() method however, the tail pointer is not set correctly. As line 210 of listing 4.2 shows, the tail pointer is being set to the array size. That number is not a valid array element however since it is outside of the array boundaries. When `Enqueue` is called, and there is at least one empty element in the storage array, the write to `object[_tail]` therefore fails. The correct value for `_tail` after `TrimToSize` is `0`.

The bug is difficult to find with mere runtime tests, since it only manifests itself if the sequence `TrimToSize`, `Dequeue`, `Enqueue` (or a variation thereof with an arbitrary number of `Dequeue` calls) is called. If `Enqueue` is called after a `TrimToSize`, the storage array is again reallocated, `_tail` is set to a valid value and the problematic condition is corrected.

```
static void Main(string[] args) {
   Queue queue = new Queue();

   queue.Enqueue(null);
   queue.TrimToSize();
   queue.Dequeue();
   queue.Enqueue(null); // throws IndexOutOfRangeException
}
```

Listing 4.1: Triggering the System.Collections.Queue Bug

```
43    public class Queue : ICollection, IEnumerable, ICloneable {
45       private object[] _array;
46       private int _head = 0; // points to the first used slot
47       private int _size = 0;
48       private int _tail = 0;
49       private int _growFactor;
51
52       public Queue () : this (32, 2.0F) {}
66       public Queue (int initialCapacity, float growFactor) {
67          if (initialCapacity < 0)
68             throw new ArgumentOutOfRangeException("capacity", "Needs a non-negative
                   number");
69          if (!(growFactor >= 1.0F && growFactor <= 10.0F))
70             throw new ArgumentOutOfRangeException("growFactor", "Queue growth
                  factor must be between 1.0 and 10.0, inclusive");
71
72          _array = new object[initialCapacity];
73
74          this._growFactor = (int)(growFactor * 100);
75       }
162
163      public virtual object Dequeue ()
164      {
166         if (_size < 1)
167            throw new InvalidOperationException ();
168         object result = _array[_head];
169         _array [_head] = null;
170         _head = (_head + 1) % _array.Length;
171         _size--;
172         return result;
173      }
174
175      public virtual void Enqueue (object obj) {
177         if (_size == _array.Length)
178            grow ();
179         _array[_tail] = obj;
180         _tail = (_tail+1) % _array.Length;
181         _size++;
183      }
203
204      public virtual void TrimToSize() {
206         object[] trimmed = new object [_size];
207         CopyTo (trimmed, 0);
208         _array = trimmed;
209         _head = 0;
210         _tail = _head + _size;
211      }
377   }
```

Listing 4.2: Parts of Mono's System.Collections.Queue. The line numbers correspond to the ones in the original file, Queue.cs as it appeared in Mono version 1.1.15.

In the Spec# version of the code I've specified the following invariant (see listing 3.11):

```
invariant _array.Length > 0 ==> _head < _array.Length;
```

Which catches this bug quickly. However, Boogie flagged the error even before I had set this invariant, as it determined that `_tail` might be used as array index despite being outside of the valid index boundaries.

4.2. Bugs in Spec#

While doing this case study, I've come across a number of bugs in Spec#. This was to be expected, given that Spec# is under heavy development, and nowhere near ready for production use. I've tried to track the bugs down when I found them and reported them to the Spec# team, unless they were already known. I'll give an overview over those bugs in this section.

There are also a number of features that are simply not yet implemented, for example the handling of floating point numbers or the bit-shift operators. I don't consider those bugs, and have not reported them to the Spec# team. Where such missing features affect this case study, I've mentioned it during the discussion of the reviewed code; I will therefore not repeat them here.

- Using the identifiers `_count`, `_exists`, `_exists1` and `_forall` did not work due to a bug in the serialization format. This has been fixed.
- Boogie produced wrong BPL code for the `Owner.Is`, `Owner.Same` and `Owner.None` functions: it writes `$Heap` when it should write `$h`. This has been fixed.
- The **`virtual`** keyword, when used with the `GetEnumerator` function of a class implementing the `IEnumerable` interface, caused the Spec# compiler to throw a CS2621 error. This has been fixed.
- The **`virtual`** keyword, when used with the `GetEnumerator` function of a class implementing the `IEnumerable` interface, currently causes the Spec# compiler to throw a CS0029 error. This is probably a follow-up bug of the previous bug.
- The `System.Array.Clear` method had a wrong precondition. It said

  ```
  requires array.Length - (index + array.GetLowerBound(0)) < length;
  ```

  when it should have said

  ```
  requires array.Length - (index + array.GetLowerBound(0)) >= length;.
  ```

  This has been fixed.
- A constructor that has not been marked as `[NotDelayed]`, but reads from **`this`**, doesn't always produce a compiler error. This has been fixed.
- Returning peer objects, ie. having functions with the attribute `[Owned("peer")]` will cause the compiler to throw a CS2692 error. This has been reported and will be fixed in a future release.

5. Conclusions

5.1. Advantages of Using Spec#

During the process of doing this case study, and in particular after finding the bug in Mono's `Queue` class (see section 4.1), I've been able to confirm my hopes that computer-assisted proving of code is an effective way to increase the correctness and thus reliability of software. Since it is up to the software developer, how much use he wishes to make of the features offered by Spec#, there are different ways of using the language:

The most basic way to use Spec# is using the Spec# compiler as a drop-in replacement for the C# compiler, This only requires minimal changes to existing C# code; for example, some calls will now use non-null types as parameters or result types, as the compiler will link the program against the Spec# version of the Microsoft core library. While this approach is easy, and the conversion done quickly, the advantages are limited; existing code barely gains any reliability. The calls to the Spec# library must now observe the Spec# preconditions, but these preconditions will have been checked explicitly in the C# version of the core library anyway. It is very likely that the C# version of the Microsoft core library is just as correct as the Spec# version: any bugs found during the translation to Spec# will have been fixed in the C# version as well. Any preconditions are only checked at runtime, so it may be possible for the program to encounter situations where core library functions are called with parameters that violate those functions' preconditions.

The next step is making use of the features that Spec# offers. Changing types to non-null types is very easy, and can completely prevent null-references. This probably won't do much though. Adding method contracts, invariants and the ownership model allows for real improvements. In my opinion, a large part of the correctness improvement comes not so much from the error checking that these features allow, but simply from the fact that they lead the software designer to a different way of thinking about the code. To write low-level specifications, he has to become aware of the environment a function runs in (the preconditions), what might be considered a successful result of the function (the postcondition and invariant), and how objects relate to each other (the ownership model). Preconditions are probably the most important part here. Formulating a precondition means realizing that there might be states where the function does not make sense, and defining the states in which it does. Since preconditions are subject to inheritance, these considerations can affect more than just a single function, which introduces a certain amount of abstraction. This alone might lead the software designer to discover flaws in existing code, and to increase the reliability of the code he produces.

Since method contracts are inherited, it is not necessary to repeat them when subclassing. This removes clutter from the code, making it more maintainable, particularly when the contracts are made part of an interface definition. At the same time, there is a danger that the programmer of the inheriting class may be unaware of details of the given specifications; good tools to extract those specifications, and to display them in an easily accessible form, are therefore required.

Using Boogie gives the largest correctness boost of the methods presented yet. By *proving* code to be correct, as opposed to testing that it works in given cases, which is what unit testing does, Boogie can not only replace a lot of unit testing, it can surpass it. When it comes to using code that has been fully equipped with Spec# contracts, such as the Microsoft core library, Boogie can make sure that this code is always used correctly, that is, in a manner that will not lead to unexpected behaviour at runtime. As I've shown in section 3, at the moment this is mostly

limited to relatively trivial behaviour, like making sure that array accesses always happen within the array's boundaries.

Is it possible to prove more? At this time, I can only speculate how complex a behaviour specification Boogie will be able to handle. The inability to properly process method calls in specifications with the current Boogie version is quite limiting. It seems likely though that such calls will work eventually, which will make it possible to prove, for example, that the element just pushed onto a stack is the one that is returned when the `Peek` method is called. Whether proving complex, non-trivial behaviour is feasible remains to be seen.

5.2. Conceptual Issues

Spec# currently has a number of conceptual issues that make using it difficult. The most obvious of these is documentation. Spec# adds many things to the C# language, both complex features and little details, some of which are undocumented, while for others the documentation is spread over papers and mailing lists. This does not come unexpected, seeing how Spec# is still in development, and neither feature-complete nor feature-stable[1], however it is an issue when getting started with Spec# now. I have been able to gain some insight from looking at the Spec# source code, however the source code made accessible to me unfortunately did not include the out-of-band contracts for the Microsoft core library, which would have been instructive.

Another issue is that of proving speed. At the moment, it is not unusual for Boogie (in conjunction with Simplify) to require minutes to prove a single, medium-complexity class even on a fast system. This seriously slows down the fast program-compile cycles that are common today. It is possible to reduce the proving time by giving methods the `[Verify(false)]` attribute, but this is user-unfriendly and of limited effectiveness. However, even if Boogie is too slow to be part of the program-compile cycle, it is fast enough to make proving code feasible at less frequent intervals.

I am unsure how this situation will change in the future. Provers are a field of active research and development, so I expect them to become faster and more efficient. Also, the processing speed of computers is growing exponentially ("Moore's Law"). Unfortunately, program complexity is growing too, and upcoming expected features of Boogie (like methods calls in method contracts) will likely introduce a speed hit. Boogie does not use threading at the moment, so it is constrained to a single CPU. Finding ways to parallelize the proving into several threads will allow it to make much better use of modern processors.

The biggest issue that I currently see however is that of language complexity. Spec#, as it is now, is considerably more difficult to understand than a common language like C#, probably too difficult for the average programmer. One part of that is because some of the features themselves are complex (like the ownership model), but the other part of it is that Boogie currently requires to programmer to use all features. It is not possible to use only some features, for example non-null types, which are very easy to understand yet useful, while omitting more advanced ones like the ownership model.

One possible way to solve this and make Spec# more accessible is to allow it for different features of Spec# to be used independently, so that it is for example possible to use only non-null types, pre- and postconditions, and to omit invariants and the ownership model. This seems to indeed be what the Spec# team plans to do. Another possibility is to restructure software engineering teams so that a team of "average programmers" is assigned a software developer that understands Spec# well and handles the proving for them, while they write the code.

[1] The way some features work in Spec# is still subject to change.

6. How to Translate C# Code to Spec#

6.1. Introduction

In this chapter I'll show how to convert a given C# program into a Spec# one. There are, unfortunately, a few caveats. First, Spec# is currently changing at a relatively rapid pace as it is being developed, so some information here might become outdated quickly. Also, I will use in my examples code that is similar to the one presented earlier in this report. For programs that use different concepts and structures, different techniques might be necessary.

I will use as an example a simplified version of the `Stack` class, `SimpleStack`. The difference between them is that I've removed many methods in `SimpleStack` which are not necessary for this tutorial. `SimpleStack`'s code is shown in appendix A.4. What remains is a minimal stack implementation whose member fields and methods include `Resize`, `Count`, `Clear`, `GetEnumerator` and the associated `Enumerator` class, `Peek`, `Pop` and `Push`.

6.2. Getting the Spec# Compiler to Compile the Code

So let's start with taking the existing file, `SimpleStack.cs`, renaming it to `SimpleStack.ssc` and compiling it with the Spec# compiler. I use this command:

```
> ssc /debug /target:library SimpleStack.ssc
```

The compiler will give a number of error messages, most of them saying that either `SimpleStack` does not implement a certain interface member, or that some methods are not pure enough. What's going on? `SimpleStack` will no longer be compiled against the standard C# core library, but against the Spec# version of it. In the Spec# version, many interfaces are slightly different, as they make use of Spec#'s features. As `SimpleStack` implements the `ICollection` and `IEnumerable` interfaces, it needs to be adapted to those differences.

Since the Spec# core library lacks documentation at the moment, I simply go by the compiler error and warning messages for now. Let's look at each error message in turn:

- `SimpleStack.ssc(42,15): error CS0535: 'System.Collections.SimpleStack' does not implement interface member 'System.Collections.ICollection.CopyTo( System.Array!, int)'`

  In the Spec# core library, the type of the first argument in `CopyTo(Array, int)` has been changed to a non-null type, `Array!` (notice the `!`, which is part of the type) so I modify `SimpleStack`'s `CopyTo` method to use a non-null type too.

- `SimpleStack.ssc(42,15): error CS0536: 'System.Collections.SimpleStack' does not implement interface member 'System.Collections.ICollection.SyncRoot. get'. 'System.Collections.SimpleStack.SyncRoot.get' is either static, not public, or has the wrong return type.`

The main cause for this error message is that the return value type has been changed to a non-null type in the Spec# version of the interface. Therefore, I change the return value type of the `SyncRoot` getter to `object!`.

- `SimpleStack.ssc(42,15): error CS0536: 'System.Collections.SimpleStack' does not implement interface member 'System.Collections.IEnumerable.GetEnumerator()'. 'System.Collections.SimpleStack.GetEnumerator()' is either static, not public, or has the wrong return type.`

  The return value type of the `GetEnumerator` method is now a non-null type too. I change its return value type to `IEnumerator!`.

- `SimpleStack.ssc(72,4): error CS2681: 'System.Collections.SimpleStack.Count.get' is not pure enough. It either overrides or implements 'System.Collections.ICollection.Count.get' which is marked as 'Microsoft.Contracts.PureAttribute'`

  The `Count` getter has been marked as `[Pure]` in the Spec# version of the `ICollection` interface. When implementing interfaces or inheriting, member attributes must be repeated. Therefore, I add the `[Pure]` attribute to the `Count` getter. As the `[Pure]` attribute has been defined in the `Microsoft.Contracts` namespace, I add a **`using`** `Microsoft.Contracts` at the beginning of the file to be able to use it.

- `SimpleStack.ssc(76,4): error CS2681: 'System.Collections.SimpleStack.IsSynchronized.get' is not pure enough. It either overrides or implements 'System.Collections.ICollection.IsSynchronized.get' which is marked as 'Microsoft.Contracts.PureAttribute'`

  Same as above for the `IsSynchronized` getter.

With these changes, I run the Spec# compiler again. Unfortunately, I get two new error message:

- `error CS0029: Cannot implicitly convert type 'SimpleStack' to 'T'`

  This is a tricky one. The error message given is unhelpful, as it does not contain a reference to a line or a method in the code. Also, I don't use `T` anywhere. So how can I fix this?

  By compiling out parts of the code, method-by-method, and recompiling, I can find the method which causes this error. It turns out to be `GetEnumerator`. Unfortunately, due to a bug in Spec#, using the **`virtual`** keyword with `GetEnumerator` doesn't work at the moment. By commenting out just that keyword, the compiler can proceed with the compilation.

- `SimpleStack.ssc(74,3): error CS2681: 'SimpleStack.SyncRoot.get' is not pure enough. It either overrides or implements 'System.Collections.ICollection.SyncRoot.get' which is marked as 'Microsoft.Contracts.PureAttribute'`

  The error I've received before for `SyncRoot` has masked another error: the `SyncRoot` getter also needs to be marked with `[Pure]`.

With these further changes, the code finally compiles.

6.3. Making Use of Non-Null Types

While the Spec# compiler compiles the code, it also gives a number of warning messages, which are all related to possible null-dereferences. Let's have a look at the first one of each type:

- `SimpleStack(53,14): warning CS2613: Conversion to 'object[]!' fails if the value is null`

  This refers to the following line of code:

```
53        Array.Copy(contents, ncontents, count);
```

  The problem here is that `Array.Copy` uses `object[]!` as type for its first parameter, but `contents` is only an object[], and therefore, possibly **`null`**. If it is null, the conversion will be aborted with an exception.

- `SimpleStack(120,15): warning CS2614: Receiver might be null (of type 'SimpleStack')`

  This warning message refers to the `Enumerator` constructor, shown here:

```
119          internal Enumerator(SimpleStack s) {
120             stack = s;
121             modCount = s.modCount;
122             current = BOF;
123          }
```

  The parameter `s` is of a possibly-null type. If it is **`null`**, then `s.modCount` would be a null-dereference, causing an exception.

- `SimpleStack(81,4): warning CS2638: Using possibly null pointer as array`

  This refers to the following line of code:

```
81           contents[i] = null;
```

  The field `contents` is of type `object[]`, which is a possibly-null type. If `contents` is **`null`**, an access to its array elements is a null-dereference and will cause an exception.

All of this warnings seem to refer to situations where a value being null could cause a problem. It is clear from the surrounding code, that in most situations, this can't actually happen. `contents` for example is being initialized in the constructor, and after that is never assigned **`null`**, so it always contains an array object. However, what if the code contained a mistake that could produce a null-dereference? I'd only find it in testing, or maybe not at all. Spec# allows me to do better: I can change my types to non-null types, which makes sure, at compilation time, that a reference cannot refer to **`null`**.

Lets look at the fields of the `SimpleStack` and `Enumerator` classes, to see where it makes sense to use non-null types. In `SimpleStack` I have the `contents` field, which clearly must not be **`null`**, so I change its type to `object[]!`. Note that this does not mean that the array elements must not be **`null`**, but only that `contents` will always refer to a valid array. Other than that, there are only **`int`** fields, which are value types and therefore can't be **`null`** anyway.

In the `Enumerator` class, the `stack` field should be changed to a non-null type, since the enumerator can only function if it has an object to operate on.

As a consequence of these two changes, I need to make two more: `ncontents`'s type in the `Resize` method needs to be changed to `object[]!` too, and the type of the `Enumerator` constructor's parameter needs to be changed to `SimpleStack!`.

These changes promptly remove all warnings, since null-dereferences can now no longer happen. If `contents` is now assigned **`null`** in error, the compiler will mark it as such at compile-time.

There is another place where using non-null types is advantageous. I previously changed the type of the first parameter of the `CopyTo` method to a non-null type. The beginning of `CopyTo` now looks like this:

```
88    public virtual void CopyTo (Array! array, int index) {
89       if (array == null) {
90          throw new ArgumentNullException("array");
91       }
```

The check for `array == null` is redundant, since the type system guarantees that `array` is never **`null`**. I can therefore remove those three lines (89-91) with no ill effect, making the code shorter and easier to read.

6.4. Observing Library Code Contracts

The code in the Spec# core library has been equipped with contracts. Let's see if `SimpleStack` actually observes them everywhere, or whether it is possible to create situations that cause unexpected runtime errors. One possibility would be to write a test suite and try to test every possible situation. A more elegant solution however is to use Boogie to prove that the contracts are always observed. I simply run Boogie on the dll file that was produced by the compiler (note that Boogie requires that the `/debug` flag was used during compilation):

```
> Boogie SimpleStack.dll
```

Boogie will give a number of error messages, which can be classified into errors related to array boundaries, errors related to peer consistency, and errors related to unsatisfied postconditions. Let's look at the the first of these.

One of the array boundary errors is this one:

```
SimpleStack.ssc(82,4): Error: Array index possibly above upper bound
```

Which refers to this code in the Clear method:

```
81       for (int i = 0; i < count; i++) {
82          contents[i] = null;
83       }
```

Boogie thinks that `i` could grow larger than the largest allowed array index. This is because it does not infer by itself that `count` can never be larger than the array length. I can state this fact in the code by writing an invariant. An invariant is a boolean expression that is supposed to be true throughout the lifetime of an object[1]. Boogie will not only use the invariant as a basis for analyzing code, it will also make sure that the invariant is never broken inadvertently.

In this case, I add the following code in class context:

```
48    invariant count <= contents.Length;
```

After running Boogie again, I notice that it no longer things that the array index is possibly above the upper bound, however, it gives me a new error message for the `Clear` method:

[1] Not quite true, actually: it is possible (and sometimes necessary) to violate the invariant by using **expose** blocks. However, it must be reestablished at the end of such **expose** blocks.

```
SimpleStack.ssc(86,3): Error: Target of field assignment might not be sufficiently
    exposed
```

This refers to the following lines of code (note that it has been shifted by 2 lines from the code in appendix A.4, as I've inserted the **invariant** and a blank line):

```
86        count = 0;
87        current = -1;
```

Now that I've mentioned count in an invariant, Boogie will no longer allow it to be updated without the object being exposed. What does this mean? To make sure that the invariant, once established, keeps on holding, Boogie does not allow any of the variables mentioned in the invariant to be changed. If such a change is necessary, it is necessary to wrap that change in an **expose** block. While an object is exposed, the invariant is not required to hold, so any changes are OK. At the end of the expose block however, the invariant is checked, and only if it is reestablished, execution (or verification) will proceed.

What I do therefore is to rewrite the two previous lines like this:

```
86        expose (this at SimpleStack) {
87           count = 0;
88           current = -1;
89        }
```

I also add the [Additive(**false**)] attribute to the Clear method. This will no longer be necessary in future Spec# versions, as non-additive exposes will be the default for virtual methods. See the discussion of [Additive] in section 2.1.

It is not necessary to include current in the expose block at this time, since it is not yet mentioned in an invariant. I will however add a clause about current to the invariant in a moment. Having "too much" code inside an **expose** block is not a problem.

With this change, the Clear method can be verified successfully. One thing is noteworthy: now that I've added count to the invariant, Boogie should give errors for every method where it's changed (for example, inside Push), since I don't have any other **expose** blocks yet, but it doesn't. This is because these methods still give error messages; Boogie will abort verification of a method after the first error, so it is possible for an error to mask others. I will have to fix the existing errors first to see the remaining ones, and keep doing this until there are no more errors.

So let's look at another Boogie error message:

```
SimpleStack.ssc(132,5): Error: Array index possibly below lower bound
```

This refers to the Current getter inside the Enumerator class, specifically, this code:

```
127           if (modCount != stack.modCount
128              || current == BOF
129              || current == EOF
130              || current > stack.count)
131           return stack.contents[current];
```

Looking through the code of the Enumerator class, I see that current is indeed assigned negative numbers sometimes, but only -1 and -2 (EOF and BOF respectively). These values are checked for with the **if**, so by the time stack.contents is accessed, current is at least 0. Boogie at the moment doesn't know that current is always at least -2, so lets add an invariant (inside the Enumerator **class**) to express this:

```
119       invariant current >= -2;
```

Running Boogie again, I notice to my surprise that the error message doesn't change, even though `current` is now guaranteed to be at least 0 after the **`if`**. To check whether Boogie knows this, I can add an assertion:

```
134         assert current >= 0;
```

Verifying the assertion promptly fails. Even after changing the assertion to

```
134         assert current >= -2 && current != -2 && current != -1 ==> current >= 0;
```

which is a tautology, verification fails. So I've run into a situation here where Boogie can't prove the code, even though it is correct. These situations are still common, given that Boogie is still under heavy development. There are ways to work around it, however: I could replace the **`assert`** keyword above with **`assume`**, which will make Boogie blindly believe the stated assumption. Or I can try to find another assertion, which will point Boogie in the right direction. This requires guesswork and trial & error most of the time, but I've been able to find such an assertion for this case:

```
134         assert current >= -1 && current != -1 ==> current >= 0;
```

Now the previous error message is gone, Boogie no longer believes that `current` could be below the lower array bound. However, I get a new, similar error message:

```
SimpleStack.ssc(135,5): Error: Array index possibly above upper bound
```

(Note that this is the same line of code as in the above error message; the line numbers have changed because of the code I've inserted in the meantime.) I can add a clause to the invariant which says that `current` will always be smaller than `stack.contents.Length`. This is not correct however: After an `Enumerator` object has been created, elements could be popped from the stack. The next time `Current` on that enumerator is accessed, it is possible that `current` is larger than `stack.contents.Length`! So what I should state instead is that `current` is smaller than `stack.contents.Length` as long as the stack object on which the enumerator operates isn't modified:

```
120       invariant modCount == stack.modCount ==> current < stack.contents.Length;
```

Unfortunately, there are problems with that at the moment. This invariant will cause errors for code in `SimpleStack`, caused by Boogie's limited ability for cross-class analysis of invariants. So I scrap that invariant for now, and instead add an assumption to the `Current` getter:

```
135             assume current < stack.contents.Length;
```

Now the previous error message is gone. There is one last error message that Boogie gives for the `Current` getter: it requires the returned object to be peer consistent. Since I don't know anything about the owners or consistency of the objects stored in the stack (I can require that they are peer consistent when pushed onto it, but they might be changed in the meantime), I decide for now to simply shut Boogie up with an assumption:

```
136             assume stack.contents[current] != null ==>
137                stack.contents[current].IsPeerConsistent;
```

The comparison against **`null`** is there so that I don't end up calling `IsPeerConsistent` on a **`null`** object. The Spec# compiler will give a warning about that anyway, but it's safe to ignore it. Now Boogie completely verifies the `Current` getter without errors.

Because of adding the invariant to `Enumerator`, Boogie now gives errors for the methods `MoveNext` and `Reset` where `current` is changed, since the **`expose`** blocks are missing. Therefore, I add the

`[Additive(false)]` attribute to those methods, and wrap the changes to `current` into **expose** blocks.

The `MoveNext` method now looks like this:

```
142       [Additive(false)]
143       public virtual bool MoveNext() {
144          if (modCount != stack.modCount)
145             throw new InvalidOperationException();
146
147          switch (current) {
148          case BOF:
149             expose (this at Enumerator) {
150                current = stack.current;
151             }
152             return current != -1;
153
154          case EOF:
155             return false;
156
157          default:
158             expose (this at Enumerator) {
159                current--;
160             }
161             return current != -1;
162          }
163       }
```

Boogie gives an error message for the end of the first **expose** block inside `MoveNext`:

```
SimpleStack.ssc(151,5): Error: Object invariant possibly does not hold: invariant
    current >= -2;
```

I've specified that `Enumerator`'s `current` is always at least -2, but so far I've said nothing about the range of values that `SimpleStack`'s `current` can have. Boogie assumes that `stack.current` might have a value lower than -2, which, when assigned to `Enumerator`'s `current`, would break `Enumerator`'s invariant. So let's look at `SimpleStack`'s current. When analyzing the code I see that it is always at least -1, and more than that, it always points to the top of stack. This gives the relation `current + 1 == count`. I decide thus to specify the following additional invariant clauses for `SimpleStack`:

```
49    invariant current >= -1;
50    invariant current + 1 == count;
```

Unfortunately, Boogie still gives the same error for `MoveNext`. Invariants are private to their containing class, so `Enumerator` doesn't have access to `SimpleStack`'s invariant. What I can do here is to repeat all three clauses of `SimpleStack`'s invariant as **assume** statements here:

```
152          assume stack.count <= stack.contents.Length;
153          assume stack.current >= -1;
154          assume stack.current + 1 == stack.count;
```

Now Boogie is happy with `MoveNext`, and with it, all of `Enumerator`.

I turn my attention the next error message, with refers to the `Resize` method:

```
SimpleStack.ssc(55,3): Error: Array size possibly negative;
```

Why does Boogie complain that the array size, `ncapacity` is possibly negative when it is set to a number of at least 16? At the moment, Boogie doesn't know what the `Math.Max` statement means. It just sees `Math.Max` as some function which returns a value about which nothing is known, other than that it is an **`int`**. I could now either say **`assume`** `ncapacity >= 16` after the `Math.Max` call, or I could replace `Math.Max` altogether with something that Boogie can understand. In this case, I decide to do the latter and change the assignment to:

```
54        ncapacity = ncapacity < 16 ? 16 : ncapacity;
```

Now **`new`** `object[ncapacity]` is valid, as far as Boogie is concerned. However there now is an error for the `Array.Copy` call two lines below:

```
SimpleStack.ssc(57,3): Error: Call of System.Array.Copy(System.Array! sourceArray,
    System.Array! destinationArray, int length), unsatisfied precondition: requires
     length <= destinationArray.GetLowerBound(0) + destinationArray.Length;
```

`Array.Copy` has several preconditions, and Boogie cannot prove that they are always observed. What is required here is that the length of `ncontents`, which is `ncapacity`, is no smaller than `count`. I can state that as a precondition of the `Resize` method:

```
52     private void Resize(int ncapacity)
53        requires ncapacity >= count;
```

This precondition can be taken for granted inside `Resize` (since, if it doesn't hold, the body of `Resize` isn't actually executed), so Boogie can now prove that the mentioned precondition of `Array.Copy` is observed.

Boogie will now give an error about another precondition of `Array.Copy` which requires that `sourceArray` be peer-consistent. To solve this (locally as we'll see), I specify another precondition for `Resize`, **`requires`** `contents.IsPeerConsistent`.

There is one remaining problem with the `Resize` method. I can't assign a new objects to contents, without exposing **`this`**. I therefore add an **`expose`** block, using a non-additive expose because `Resize` will be called from virtual methods in **`this`**:

```
52     [Additive(false)]
53     private void Resize(int ncapacity)
54        requires ncapacity >= count;
55        requires contents.IsPeerConsistent;
56     {
57        ncapacity = ncapacity < 16 ? 16 : ncapacity;
58        object[]! ncontents = new object[ncapacity];
59
60        Array.Copy(contents, ncontents, count);
61
62        capacity = ncapacity;
63        expose (this at SimpleStack) { contents = ncontents; }
64     }
```

The next error message is:

```
Object returned by method SimpleStack.SyncRoot.get must be peer consistent
```

Since SyncRoot simply returns **`this`**, The easy solution here would be to add `IsPeerConsistent` as a precondition of the getter. Unfortunately, that doesn't work, since `SyncRoot` is an interface method implementation, which means that additional preconditions are not allowed (and this precondition apparently is not given in the interface definition). What remains is using an assume statement:

```
80    public virtual object! SyncRoot {
81       [Pure] get {
82          assume IsPeerConsistent;
83          return this;
84       }
85    }
```

On to the next error message:

```
SimpleStack.ssc(113,4): Error: Array index possibly below lower bound
```

This refers to the array access inside this **for** loop in the CopyTo method:

```
112       for (int i = current; i != -1; i--) {
113          array.SetValue(contents[i],
114                  count - (i + 1) + index);
115       }
```

Boogie is still quite bad at inferring properties of objects and values inside loops. In this case, it does realize that, inside the loop, `i` is at least 0 when accessing `contents[i]`. This follows from the invariant, which specifies that `current` is always equal to or larger than -1, and the loop condition, which prevents the loop body from being executed if `i` is -1. I can help Boogie out by specifying a loop invariant which expresses that `i >= -1`. Why not `i >= 0`? Loop invariants have to hold after the last iteration of the loop, when the loop condition is false. This loop invariant is already enough, together with the loop condition, Boogie is able to prove that `i` is at least 0 inside the loop.

```
112       for (int i = current; i != -1; i--)
113          invariant i >= -1
114       {
115          array.SetValue(contents[i],
116                  count - (i + 1) + index);
117       }
```

Rerunning Boogie, I get a similar error message for the same code:

```
SimpleStack.ssc(114,4): Error: Array index possibly above lower bound
```

Boogie also can't infer that `i` is always smaller than `contents`' length. This I can easily specify by extending the loop invariant:

```
112       for (int i = current; i != -1; i--)
113          invariant i >= -1 && i <= current;
114       {
115          array.SetValue(contents[i],
116                  count - (i + 1) + index);
117       }
```

Still at the same line, Boogie now gives the following error message:

```
SimpleStack.ssc(115,4): Error: Call of System.Array.SetValue(object value, int
    index), unsatisfied precondition: requires index <= this.GetUpperBound(0);
```

This is exactly the condition that is being checked with the preceding **if**. I can add it to the loop invariant as well.

Boogie will now give one last error message for `CopyTo`, again in the same place:

```
SimpleStack.ssc(116,4): Error: Call of 'System.Array.SetValue(object value, int
    index)': the object passed as value of parameter 'value' must be peer
    consistent
```

The parameter in question is `contents[i]`. Unfortunately, I don't know anything about the ownership of the objects stored in the stack, so all I can do at this time is give an assumption, saying `contents[i].IsPeerConsistent`, unless it is **`null`**.

```
112       for (int i = current; i != -1; i--)
113          invariant i >= -1 && i <= current;
114          invariant count <= array.Length - index;
115       {
116          assume contents[i] != null ==> contents[i].IsPeerConsistent;
117          array.SetValue(contents[i],
118                  count - (i + 1) + index);
119       }
```

As before, this will prompt the Spec# compiler to display another warning with the intention of making me aware of the fact that `contents[i]` might be **`null`** when I access `IsPeerConsistent` on it, but of course this is bogus.

With the next error message I have the same problem:

```
Object returned by method SimpleStack.Peek() must be peer consistent
```

Peek returns `contents[current]`. Again, I can do nothing but specify the assumption that if `contents[current]` is not **`null`**, it is peer consistent.

The next error message refers to the change of `count` in `Pop`:

```
SimpleStack(214,4): Error: Target object of field assignment might not be
    sufficiently exposed
```

Since I've made `count` part of the invariant, I need to expose **`this`** when changing it. Because `Pop` is a virtual method, I need to use a non-additive expose, so I also add `[Additive(false)]` as `Pop`'s attribute. While here, I do the same for `Push`.

Next I get errors for the two `Resize` calls in `Pop` and `Push`:

```
SimpleStack(226,6): Error: Call of SimpleStack.Resize(int ncapacity), unsatisfied
    precondition: requires contents.IsPeerConsistent;
SimpleStack(240,5): Error: Call of SimpleStack.Resize(int ncapacity), unsatisfied
    precondition: requires contents.IsPeerConsistent;
```

I'm not sure why I get these. As I haven't changed `contents` inside those methods, it is still consistent, and, because it has no peers (as `SimpleStack` contains no other owned objects), also peer consistent. I therefore give two more assumptions saying that `contents.IsPeerConsistent`.

The next error message is the similar to the ones I've seen before:

```
Object returned by method SimpleStack.Pop() must be peer consistent
```

The fix is the same as well.

The next error message is more interesting:

```
SimpleStack.ssc(248,3): Error: Object invariant possibly does not hold:
    invariant count <= contents.Length;
```

This refers to the end of the **expose** block in Push. Boogie says that the invariant might be violated because count is incremented, which might put it above contents.Length. This clearly can't happen, because the array size is doubled with the Resize call if count is already equal to contents.Length.

Possible solutions would be to adding a clause to the invariant which says that capacity is equal to contents.Length, and adding a postcondition to Resize saying that contents.Length is now at least as large as the argument given. Unfortunately, Boogie fails here; even when doing these things, the error message remains. Therefore, the only option is to give another assumption:

```
245       assume count < contents.Length;
```

The next error message I can fix properly. It says:

```
SimpleStack.ssc(242,3): Error: RHS might not be a subtype of the element type of
    the array being assigned
```

This error refers to the assignment at the end of Push. It is a result of the covariance of C#/Spec# which allows an array declared as object[] to actually be an array of any type. If contents were to be an array of a type other than object however, assigning an object to it would fail. What I need to do here is to specify that contents will always only contain object arrays. I do this by adding the following invariant clause to SimpleStack:

```
51    invariant contents.GetType() == typeof(object[]);
```

Now Push is verified and deemed correct.

The next error message is:

```
Method SimpleStack.System.Collections.ICollection.Icollection.get_Count(),
    unsatisfied postcondition: ensures result >= 0;
```

Since the Count getter simply returns count, I can extend the existing invariant clause for it. I've already specified the upper limit for count, and it makes sense (and is required by the postcondition mentioned in above error message) to specify a lower limit too. I therefore change the invariant clause to:

```
48    invariant count >= 0 && count <= contents.Length;
```

This allows me to repeat the postcondition in the Count implementation, which now looks like this:

```
73    public virtual int Count {
74       [Pure] get
75          ensures result >= 0;
76       {
77          return count;
78       }
79    }
```

Now Boogie is able to prove Count to be correct.

I have now two error messages from Boogie which remain:

```
Method SimpleStack.System.Collections.IEnumerable.IEnumerable.GetEnumerator(),
    unsatisfied postcondition: ensures Owner.Same(this, result);
Object returned by method SimpleStack.Systems.Collections.ICollection.ICollection.
    get_SyncRoot() must be peer consistent
```

The first of these I can't fix, as it is caused by a bug in Spec#, as discussed earlier. I would have to specify the attribute `[Owned("peer")]` for the `GetEnumerator` method, but unfortunately, this causes other errors. So for now, I ignore it.

The second error is related to one I've received before. I already gave the assumption in the `SyncRoot` getter saying that the returned object, **`this`**, is peer consistent. By also specifying this as a postcondition, Boogie accepts this properly:

```
85    public virtual object! SyncRoot {
86       [Pure] get
87          ensures result.IsPeerConsistent
88       {
89          assume IsPeerConsistent;
90          return this;
91       }
92    }
```

So what have I achieved now? By giving the code specifications (and the occasional assumption), I can have Boogie prove that all calls to to the Spec# core library are correct, that is, they observe the specifications given for it. This means, for example, that no matter how the code is abused, I will never see any `IndexOutOfRangeExceptions` when accessing arrays. If the code were incorrect, Boogie would produce errors that I would not be able to "fix" by adding specification (I could of course add assumptions to silence Boogie, but then I would not be able to explain why those assumptions are valid).

6.5. Creating My Own Contracts

I can do more. Using Spec#, I can show that not only does `SimpleStack` use the core library correctly, `SimpleStack` itself also does what it claims it does. But... what exactly does `SimpleStack` claim to do? So far, not much: the code is completely undocumented, and the only hint as to its functionality is given through the name of the class and its methods. Let's change that by adding specifications. Once I have these, I can use Boogie to prove that the code does behave in accordance with the specifications, or, in other words, that it does what it claims it does.

When adding specifications, I can mostly ignore methods I inherit or which are given by interfaces, as I can't specify additional preconditions for them. Having proper specifications for those methods is something that needs to be done in the context of the original base class, or the interface. In `SimpleStack`, there are only five methods which are "new", that is, neither inherited nor part of interfaces: `Clear`, `Peek`, `Pop`, `Push` and `Resize`. Last but not least, there is the constructor.

Clear

Let's start with `Clear`. `Clear` removes all elements for `SimpleStack`'s storage array and sets `count` to 0, and `current` to -1. Here's the code:

```
94    [Additive(false)]
95    public virtual void Clear() {
96       modCount++;
97
98       for (int i = 0; i < count; i++) {
99          contents[i] = null;
100      }
101
```

```
102         expose(this at SimpleStack) {
103             count = 0;
104             current = -1;
105         }
106     }
```

An easy way to document that `Clear` empties the stack is to say that after `Clear`, `Count` will be 0. Unfortunately, accessing `Count` really means a method call (to `Count`'s getter), and Boogie doesn't currently support method calls in method contracts. However, there is an alternative: Since the `Count` getter only returns the private field `count`, I can just refer to that instead. To be able to refer to private fields in public method contracts, I need to mark them as `[SpecPublic]`. The beginning of the `Count` method now looks like this:

```
94      [Additive(false)]
95      public virtual void Clear()
96          ensures count == 0;
97      {
```

This postcondition clearly, if very concisely, documents what the `Clear` method does. To be consistent and create the connection between `count` and `Count` (which is documented in the `ICollection` interface), I also add the following postcondition to the `Count` getter:

```
76      ensures result == count;
```

Peek

The `Peek` method currently looks like this:

```
206     public virtual object Peek() {
207         if (current == -1) {
208             throw new InvalidOperationException();
209         } else {
210             assume contents[current] != null ==> contents[current].IsPeerConsistent;
211             return contents[current];
212         }
213     }
```

The `Peek` method returns the last object pushed onto the stack, without modifying the stack. The first part is difficult to express in the context of `Peek`: how exactly do I specify "the last object pushed onto the stack"? Making more internal variables `[SpecPublic]` is not helpful: while I could express that `Peek` always returns the array element with the largest index, it is not clear that there is a correlation between the order of array elements and the order of stack elements. Moreover, if the internal representation is changed to another data structure, the external specification would have to be changed too. Clearly, this should be avoided. Instead, `Peek` can be defined in terms of the other stack functions, `Pop` and `Push`. Therefore, I postpone the specification of this part of `Peek` for a moment.

The second part of `Peek`'s behaviour, ie. that it doesn't change the stack can be expressed easily: I simply mark the method as `[Pure]`.

There is another thing about `Peek` that the user of `SimpleStack` should know about: He can't call `Peek` if there are no elements on the stack. This condition is currently checked for with the **`if`**. If there are no elements on the stack, an `InvalidOperationException` is thrown. This check can be easily turned into a precondition by requiring that `count` be larger than 0 (which implies that

current is larger or equal to 0 because of the invariant), which automatically also documents this requirement.

With these changes, Peek's code now is:

```
206    [Pure]
207    public virtual object Peek()
208       requires count > 0 otherwise InvalidOperationException;
209    {
210       assume contents[current] != null ==> contents[current].IsPeerConsistent;
211       return contents[current];
212    }
```

As a side effect of these changes, the code has become shorter and easier to read, which is always welcome. From a functional point of view, it is still the same. What is Boogie able to prove with these changes? First, it will prove that the restrictions given by the [Pure] attribute, ie. that **this** isn't observably changed, are obeyed. Second, if SimpleStack is used in other code, and that code is verified with Boogie, it will prove that Peek is never called on an empty stack (and give errors if it can't).

Pop

Next I'll have a look at Pop. It currently looks like this:

```
214    [Additive(false)]
215    public virtual object Pop() {
216       if (current == -1) {
217          throw new InvalidOperationException();
218       } else {
219          modCount++;
220
221          object ret = contents[current];
222          contents [current] = null;
223
224          expose (this at SimpleStack) {
225             count--;
226             current--;
227          }
228
229          // if we're down to capacity/4, go back to a
230          // lower array size.  this should keep us from
231          // sucking down huge amounts of memory when
232          // putting large numbers of items in the Stack.
233          // if we're lower than 16, don't bother, since
234          // it will be more trouble than it's worth.
235          if (count <= (capacity/4) && count > 16) {
236             assume contents.IsPeerConsistent;
237             Resize(capacity/2);
238          }
239
240          assume ret != null ==> ret.IsPeerConsistent;
241          return ret;
242       }
243    }
```

Pop removes the topmost element from the stack and returns it. This implies three things: The element that is returned is the same as the one that used to be returned by Peek before, it defines the "topmost element" as "the element returned by Peek", and last, the sentence says that the stack's element count is decreased by one. All of this can be specified as a precondition.

As does Peek, Pop requires that the stack not be empty when called. I specify the same precondition that I have for Peek and remove the **if**. This is Pop after these changes:

```
214    [Additive(false)]
215    public virtual object Pop()
216       requires count > 0 otherwise InvalidOperationException;
217       ensures result == old(Peek());
218       ensures count == old(count) - 1;
219    {
220       modCount++;
221
222       object ret = contents[current];
223       contents [current] = null;
224
225       expose (this at SimpleStack) {
226          count--;
227          current--;
228       }
229
230       // if we're down to capacity/4, go back to a
231       // lower array size.  this should keep us from
232       // sucking down huge amounts of memory when
233       // putting large numbers of items in the Stack.
234       // if we're lower than 16, don't bother, since
235       // it will be more trouble than it's worth.
236       if (count <= (capacity/4) && count > 16) {
237          assume contents.IsPeerConsistent;
238          Resize(capacity/2);
239       }
240
241       assume ret != null ==> ret.IsPeerConsistent;
242       return ret;
243    }
```

Unfortunately, since method calls in specifications don't work (yet), Boogie can't actually prove that **result** == **old**(Peek()), but instead gives an error message. So for now I comment out line 217 to have it at least verify the rest of the method. I expect this postcondition to work in future Spec# versions though.

Boogie will prove that count decreases, and it will also prove that code that uses SimpleStack doesn't call Pop on an empty stack.

Push

The Push method currently looks like this:

```
245    [Additive(false)]
246    public virtual void Push(Object o) {
247       modCount++;
248
```

```
249         if (capacity == count) {
250             assume contents.IsPeerConsistent;
251             Resize(capacity * 2);
252         }
253
254         assume count < contents.Length;
255
256         expose (this at SimpleStack) {
257             count++;
258             current++;
259         }
260
261         contents[current] = o;
262     }
```

Push adds a new element to the top of the stack. In other words, it increases the count of elements by one, and the added element is the one that will now be returned by Peek. Since there is no defined upper limit on the number of elements on the stack, Push can at least theoretically always be called. There is therefore no precondition, and all I need to add are the two postconditions. The Push method now starts like this:

```
245     [Additive(false)]
246     public virtual void Push(Object o)
247         ensures o == Peek();
248         ensures count == old(count) + 1;
249     {
```

Again, the postcondition `o == Peek()` can't actually be proved by Boogie at the moment, so I comment it out. The other postcondition is proved to hold.

Resize

Even though Resize is private, it makes sense to add specifications to it. They act as internal documentation by recording design decisions, and allow checking whether these design decisions are being followed.

Resize currently looks like this:

```
53     [Additive(false)]
54     private void Resize(int ncapacity)
55         requires ncapacity >= count;
56         requires contents.IsPeerConsistent;
57     {
58         ncapacity = ncapacity < 16 ? 16 : ncapacity;
59         object[]! ncontents = new object[ncapacity];
60
61         Array.Copy(contents, ncontents, count);
62
63         capacity = ncapacity;
64         expose (this at SimpleStack) { contents = ncontents; }
65     }
```

An important part of Resize is that it doesn't change the stack. I can express this in two preconditions: One, the total count of stack elements remains the same, and two, every element on the

stack remains the same. For this I use, for the first time in this class, a `forall` quantifier. `Resize` now starts like this:

```
53    [Additive(false)]
54    private void Resize(int ncapacity)
55       requires ncapacity >= count;
56       requires contents.IsPeerConsistent;
57       ensures count == old(count);
58       ensures forall { int i in (0:count - 1), contents[i] == old(contents[i]) };
59    {
```

Boogie successfully proves that both postconditions are observed, so the code is correct.

Constructor

Last, I have the very simple constructor:

```
69    public SimpleStack ()
70    {
71       contents = new object[default_capacity];
72       capacity = default_capacity;
73    }
```

Since the constructor takes no parameters, there are no preconditions that would make sense. However, there is one important postcondition that should be specified: after creating a stack, it is empty (as opposed to, say, filled with random data), or in other words, the element count is 0. With this postcondition, the constructor becomes:

```
69    public SimpleStack ()
70       ensures count == 0;
71    {
72       contents = new object[default_capacity];
73       capacity = default_capacity;
74    }
```

And of course, Boogie is able to prove that the postcondition holds.

Other Design Decisions

There are a number of other design decisions that could be specified. For example, I could add an invariant saying that `capacity` is never larger than `contents.Length`. Doing this makes sense, as `capacity` is in various places in the code for storage calculation, which could potentially contain errors. After adding capacity to the invariant, I would also have to enclose modifications to it in **`expose`** blocks.

I could also add **`assert`** statements in various places. The difference to other languages that also have assertions is that Boogie will try to prove that the assertions hold.

6.6. Summary

I've shown in this tutorial now how to turn a C# program into a Spec# one and have it proved correct. These are the steps I've taken:

1. Make the C# code compile with the Spec# compiler by adding non-null types and attributes as dictated by the inheritance, interface implementation and method call rules.
2. Add non-null types whereever it makes sense in order to more easily catch null-dereference errors.
3. Make sure the code observes the specifications of inherited and called methods, as well as the methods defined in interfaces by adding enough specification so that Boogie can prove it. This ensures that third-party code is used correctly.
4. Add specifications for the rest of the code to ensure the correctness of the implementation, as well as correct use of the code.

Please see appendix A.5 for the full Spec# version of `SimpleStack`.

A. Code

Comments starting with //! are comments added by me. Other comments are part of the original Mono code. Only Spec# language constructs have been added, the existing code has not been modified except where absolutely necessary; in such cases this has been noted in the comments.

A.1. BitArray

```
1  //
2  // Bit Array.cs
3  //
4  // Authors:
5  // Ben Maurer (bmaurer@users.sourceforge.net)
6  //
7  // (C) 2003 Ben Maurer
8  //
9
10 //
11 // Copyright (C) 2004 Novell, Inc (http://www.novell.com)
12 //
13 // Permission is hereby granted, free of charge, to any person obtaining
14 // a copy of this software and associated documentation files (the
15 // "Software"), to deal in the Software without restriction, including
16 // without limitation the rights to use, copy, modify, merge, publish,
17 // distribute, sublicense, and/or sell copies of the Software, and to
18 // permit persons to whom the Software is furnished to do so, subject to
19 // the following conditions:
20 //
21 // The above copyright notice and this permission notice shall be
22 // included in all copies or substantial portions of the Software.
23 //
24 // THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
25 // EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
26 // MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
27 // NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
       BE
28 // LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
29 // OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
30 // WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
31 //
32
33 using System;
34 using System.Runtime.InteropServices;
35 using Microsoft.Contracts;
36
37 namespace System.Collections {
```

```
38 #if NET_2_0
39    [ComVisible(true)]
40 #endif
41    [Serializable]
42    public sealed class BitArray : ICollection, ICloneable {
43       //! initialisation necessary because of not-delayed constructors.
44       //! old code was: int [] m_array;
45       int []! m_array = new int[0];
46       [SpecPublic] int m_length;
47       int _version = 0;
48
49       invariant m_length >= 0;
50       invariant m_array.Length >= (m_length + 31) / 32;
51       invariant m_array != null;
52
53 #region Constructors
54       [NotDelayed]
55       public BitArray (BitArray! bits)
56          requires bits.IsConsistent;
57       {
58          //! if (bits == null)
59          //!     throw new ArgumentNullException ("bits");
60
61          m_length = bits.m_length;
62          m_array = new int [(m_length + 31) / 32];
63
64          assume bits.m_array.IsPeerConsistent; //! follows from precondition
65          Array.Copy(bits.m_array, m_array, m_array.Length);
66       }
67
68       [NotDelayed]
69       public BitArray (bool []! values)
70       {
71          //! if (values == null)
72          //!     throw new ArgumentNullException ("values");
73
74          m_length = values.Length;
75          m_array = new int [(m_length + 31) / 32];
76          assume m_array.Length == (m_length + 31) / 32;
77
78          //! Added temp vars to express that m_length and m_array.Length remain
79          //! constant through the for loop.
80          int temp1 = m_length;
81          int temp2 = m_array.Length;
82
83          for (int i = 0; i < values.Length; i++)
84             invariant temp1 == m_length;
85             invariant temp2 == m_array.Length;
86          {
87             this [i] = values [i];
88          }
89       }
90
91       [NotDelayed]
```

```
92        public BitArray (byte []! bytes)
93        {
94           //! if (bytes == null)
95           //!     throw new ArgumentNullException ("bytes");
96
97           m_length = bytes.Length * 8;
98           m_array = new int [(m_length + 31) / 32];
99           assume m_array.Length == (m_length + 31) / 32;
100
101          //! Added temp vars to express that m_length and m_array.Length remain
102          //! constant through the for loop.
103          int temp1 = m_length;
104          int temp2 = m_array.Length;
105
106          for (int i = 0; i < bytes.Length; i++)
107             invariant temp1 == m_length;
108             invariant temp2 == m_array.Length;
109          {
110             setByte (i, bytes [i]);
111          }
112       }
113
114       [NotDelayed]
115       public BitArray (int []! values)
116       {
117          //! if (values == null)
118          //!     throw new ArgumentNullException ("values");
119
120          int arrlen = values.Length;
121          m_length = arrlen*32;
122          m_array = new int [arrlen];
123          Array.Copy (values, m_array, arrlen);
124       }
125
126       [NotDelayed]
127       public BitArray (int length)
128          requires length >= 0 otherwise ArgumentOutOfRangeException;
129       {
130          //! if (length < 0)
131          //!     throw new ArgumentOutOfRangeException ("length");
132
133          m_length = length;
134          m_array = new int [(m_length + 31) / 32];
135       }
136
137       [NotDelayed]
138       public BitArray (int length, bool defaultValue) : this (length)
139          requires length >= 0;
140       {
141          if (defaultValue) {
142             for (int i = 0; i < m_array.Length; i++)
143             m_array[i] = ~0;
144          }
145       }
```

```
146
147         private BitArray (int []! array, int length)
148            requires length >= 0;
149            requires array.Length >= (length + 31) / 32;
150         {
151            m_array = array;
152            m_length = length;
153         }
154   #endregion
155   #region Utility Methods
156
157         [Pure]
158         byte getByte (int byteIndex)
159            requires byteIndex >= 0 && byteIndex < (m_length + 7) / 8;
160         {
161            int index = byteIndex / 4;
162            int shift = (byteIndex % 4) * 8;
163
164            int theByte = m_array [index] & (0xff << shift);
165
166            return (byte)((theByte >> shift) & 0xff);
167         }
168
169         void setByte (int byteIndex, byte value)
170            requires byteIndex >= 0 && byteIndex < m_length / 8;
171         {
172            int index = byteIndex / 4;
173            int shift = (byteIndex % 4) * 8;
174
175            // clear the byte
176            m_array [index] &= ~(0xff << shift);
177            // or in the new byte
178            m_array [index] |= value << shift;
179
180            _version++;
181         }
182
183         [Pure]
184         void checkOperand (BitArray! operand)
185            requires operand.m_length == m_length otherwise ArgumentException;
186         {
187            //! not necessary because operand is non-nullable.
188            //! if (operand == null)
189            //!    throw new ArgumentNullException ();
190
191            //! if (operand.m_length != m_length)
192            //!    throw new ArgumentException ();
193         }
194   #endregion
195
196         public int Count {
197            [Pure]
198            get
199               ensures result == m_length;
```

```
200         { return m_length; }
201      }
202
203      public bool IsReadOnly {
204         get { return false; }
205      }
206
207      public bool IsSynchronized {
208         [Pure]
209         get { return false; }
210      }
211
212      public bool this [int index] {
213         [Pure]
214         get
215            requires index >= 0 && index < m_length otherwise
                   ArgumentOutOfRangeException;
216            requires IsPeerConsistent;
217         {
218            return Get (index);
219         }
220         set
221            requires index >= 0 && index < m_length otherwise
                   ArgumentOutOfRangeException;
222         {
223            Set (index, value);
224         }
225      }
226
227      public int Length {
228         [Pure]
229         get { return m_length; }
230         set
231            requires value >= 0 otherwise ArgumentOutOfRangeException;
232         {
233            //! if (value < 0)
234            //!     throw new ArgumentOutOfRangeException ();
235
236            int newLen = value;
237            if (m_length != newLen) {
238               int numints = (newLen + 31) / 32;
239               int [] newArr = new int [numints];
240               int copylen = (numints > m_array.Length) ? m_array.Length : numints;
241               expose (this) {
242                  assume m_array.IsPeerConsistent;
243                  Array.Copy (m_array, newArr, copylen);
244
245                  // set the internal state
246                  m_array = newArr;
247                  m_length = newLen;
248                  _version++;
249               }
250            }
251         }
```

```
252         }
253
254         public object! SyncRoot {
255            [Pure]
256            get { return this; }
257         }
258
259         public object! Clone ()
260         {
261            // LAMESPEC: docs say shallow, MS makes deep.
262            return new BitArray (this);
263         }
264
265         public void CopyTo (Array! array, int index)
266            requires index >= 0 otherwise ArgumentOutOfRangeException;
267            requires array.Rank == 1 otherwise ArgumentException;
268            requires index < array.Length otherwise ArgumentException;
269            requires array is bool[]! || array is byte[]! || array is int[]!;
270            requires array is bool[]! ==> array.Length - index >= m_length;
271            requires array is byte[]! ==> array.Length - index >= (m_length + 7) / 8;
272            requires array is int[]! ==> index + (m_length + 31) / 32 <= array.Length;
273         {
274            //! if (array == null)
275            //!    throw new ArgumentNullException ("array");
276            //! if (index < 0)
277            //!    throw new ArgumentOutOfRangeException ("index");
278            //! if (array.Rank != 1)
279            //!    throw new ArgumentException ("array", "Array rank must be 1");
280            //! if (index >= array.Length)
281            //!    throw new ArgumentException ("index", "index is greater than array.Length");
282
283            // in each case, check to make sure enough space in array
284            if (array is bool []!) {
285               //! if (array.Length - index < m_length)
286               //!    throw new ArgumentException ();
287
288               bool []! barray = (bool []!) array;
289
290               // Copy the bits into the array
291               for (int i = 0; i < m_length; i++)
292                  invariant array.Length - index >= m_length; //! precondition
293               {
294                  assume i < m_length; //! loop stop condition
295                  barray[index + i] = this [i];
296               }
297            } else if (array is byte []!) {
298               int numbytes = (m_length + 7) / 8;
299
300               //! if ((array.Length - index) < numbytes)
301               //!    throw new ArgumentException ();
302
303               byte []! barray = (byte []!) array;
304               // Copy the bytes into the array
305               for (int i = 0; i < numbytes; i++)
```

```
306                invariant numbytes == (m_length + 7) / 8;
307             {
308                assume i < numbytes; //! loop stop condition
309                barray [index + i] = getByte (i);
310             }
311          } else if (array is int []!) {
312             assume m_array.IsPeerConsistent;
313             Array.Copy (m_array, 0, array, index, (m_length + 31) / 32);
314          //! } else {
315          //!     throw new ArgumentException ("array", "Unsupported type");
316          }
317       }
318
319       public BitArray Not ()
320       {
321          int ints = (m_length + 31) / 32;
322          assert ints <= m_array.Length;
323          for (int i = 0; i < ints; i++) {
324             assume i < ints && ints <= m_array.Length;
325             m_array [i] = ~m_array [i];
326          }
327
328          _version++;
329          return this;
330       }
331
332       public BitArray And (BitArray! value)
333          requires value.m_length == m_length otherwise ArgumentException;
334       {
335          checkOperand (value);
336
337          int ints = (m_length + 31) / 32;
338          assert ints <= m_array.Length && ints <= value.m_array.Length;
339          for (int i = 0; i < ints; i++)
340          {
341             assume i < ints; //! loop stop condition
342             assume ints <= m_array.Length && ints <= value.m_array.Length; //!
                    asserted
343             m_array [i] &= value.m_array [i];
344          }
345
346          _version++;
347          return this;
348       }
349
350       public BitArray Or (BitArray! value)
351          requires value.m_length == m_length otherwise ArgumentException;
352       {
353          checkOperand (value);
354
355          int ints = (m_length + 31) / 32;
356          assert ints <= m_array.Length && ints <= value.m_array.Length;
357          for (int i = 0; i < ints; i++) {
358             assume i < ints; //! loop stop condition
```

```
359             assume ints <= m_array.Length && ints <= value.m_array.Length; //!
                    asserted
360             m_array [i] |= value.m_array [i];
361          }
362
363          _version++;
364          return this;
365       }
366
367       public BitArray Xor (BitArray! value)
368          requires value.m_length == m_length otherwise ArgumentException;
369       {
370          checkOperand (value);
371
372          int ints = (m_length + 31) / 32;
373          assert ints <= m_array.Length && ints <= value.m_array.Length;
374          for (int i = 0; i < ints; i++) {
375             assume i < ints; //! loop stop condition
376             assume ints <= m_array.Length && ints <= value.m_array.Length; //!
                    asserted
377             m_array [i] ^= value.m_array [i];
378          }
379
380          _version++;
381          return this;
382       }
383
384       [Pure]
385       public bool Get (int index)
386          requires index >= 0 && index < m_length otherwise
                ArgumentOutOfRangeException;
387       {
388          //! if (index < 0 || index >= m_length)
389          //!     throw new ArgumentOutOfRangeException ();
390
391          assume index >= 0 ==> (index >> 5) >= 0;
392          assume (index >> 5) == (index / 32);
393          return (m_array [index >> 5] & (1 << (index & 31))) != 0;
394       }
395
396       public void Set (int index, bool value)
397          requires index >= 0 && index < m_length otherwise
                ArgumentOutOfRangeException;
398       {
399          //! if (index < 0 || index >= m_length)
400          //!     throw new ArgumentOutOfRangeException ();
401
402          assume index >= 0 ==> (index >> 5) >= 0;
403          assume (index >> 5) == (index / 32);
404          if (value)
405             m_array [index >> 5] |= (1 << (index & 31));
406          else
407             m_array [index >> 5] &= ~(1 << (index & 31));
408
```

```
409            _version++;
410         }
411
412         public void SetAll (bool value)
413         {
414            if (value) {
415               for (int i = 0; i < m_array.Length; i++)
416                  m_array[i] = ~0;
417            }
418            else {
419               //! XXX this is an erroneous precondition in System.Array.Clear
420               assume m_array.Length < m_array.Length;
421               Array.Clear (m_array, 0, m_array.Length);
422            }
423
424            _version++;
425         }
426
427         public IEnumerator! GetEnumerator ()
428         {
429            return new BitArrayEnumerator (this);
430         }
431
432         [Serializable]
433         class BitArrayEnumerator : IEnumerator, ICloneable {
434            BitArray! _bitArray;
435            bool _current;
436            int _index, _max, _version;
437
438            invariant _index >= -1;
439
440            public object! Clone () {
441               return MemberwiseClone ();
442            }
443
444            public BitArrayEnumerator (BitArray! ba)
445            {
446               _index = -1;
447               _bitArray = ba;
448               _max = ba.m_length;
449               _version = ba._version;
450            }
451
452            public object Current {
453               get {
454                  assume _bitArray.IsPeerConsistent;
455                  if (_index == -1)
456                     throw new InvalidOperationException ("Enum not started");
457                  if (_index >= _bitArray.Count)
458                     throw new InvalidOperationException ("Enum Ended");
459
460                  return _current;
461               }
462            }
```

```
463
464          public bool MoveNext ()
465          {
466             checkVersion ();
467
468             assume _bitArray.IsPeerConsistent;
469             if (_index < (_bitArray.Count - 1)) {
470                expose (this) {
471                   assume _bitArray.IsPeerConsistent;
472                   _current = _bitArray [++_index];
473                }
474                return true;
475             }
476             else
477                expose (this) {
478                   assume _bitArray.IsPeerConsistent;
479                   _index = _bitArray.Count;
480                }
481
482             return false;
483          }
484
485          public void Reset ()
486          {
487             checkVersion ();
488             expose (this) {
489                _index = -1;
490             }
491          }
492
493          void checkVersion ()
494          {
495             if (_version != _bitArray._version)
496                throw new InvalidOperationException ();
497          }
498       }
499    }
500 }
```

A.2. Queue

```
1  //
2  // System.Collections.Queue
3  //
4  // Author:
5  //     Ricardo Fernández Pascual
6  //
7  // (C) 2001 Ricardo Fernández Pascual
8  //
9
10 //
11 // Copyright (C) 2004 Novell, Inc (http://www.novell.com)
12 //
13 // Permission is hereby granted, free of charge, to any person obtaining
14 // a copy of this software and associated documentation files (the
15 // "Software"), to deal in the Software without restriction, including
16 // without limitation the rights to use, copy, modify, merge, publish,
17 // distribute, sublicense, and/or sell copies of the Software, and to
18 // permit persons to whom the Software is furnished to do so, subject to
19 // the following conditions:
20 //
21 // The above copyright notice and this permission notice shall be
22 // included in all copies or substantial portions of the Software.
23 //
24 // THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
25 // EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
26 // MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
27 // NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
       BE
28 // LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
29 // OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
30 // WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
31 //
32
33 using System;
34 using System.Collections;
35 using System.Runtime.InteropServices;
36 using Microsoft.Contracts;
37
38 namespace System.Collections {
39
40 #if NET_2_0
41    [ComVisible(true)]
42 #endif
43    [Serializable]
44    public class Queue : ICollection, IEnumerable, ICloneable {
45
46       private object[]! _array;
47       private int _head = 0; // points to the first used slot
48       [SpecPublic] private int _size = 0;
49       private int _tail = 0;
50       private int _growFactor;
```

```
51        private int _version = 0;
52
53        invariant _head >= 0;
54        invariant _array.Length > 0 ==> _head < _array.Length;
55        invariant _array.Length == 0 ==> _head == 0;
56        invariant _tail >= 0;
57        invariant _array.Length > 0 ==> _tail < _array.Length;
58        invariant _array.Length == 0 ==> _tail == 0;
59        invariant _size >= 0 && _size <= _array.Length;
60        invariant _array.GetType() == typeof(object[]);
61
62        public Queue () : this (32, 2.0F) {}
63        public Queue (int initialCapacity) : this (initialCapacity, 2.0F)
64           requires initialCapacity >= 0;
65        {}
66        public Queue(ICollection! col) : this (col == null ? 32 : col.Count)
67           requires col.IsPeerConsistent;
68        {
69           //! if (col == null)
70           //!     throw new ArgumentNullException ("col");
71
72           // We have to do this because msft seems to call the
73           // enumerator rather than CopyTo. This affects classes
74           // like bitarray.
75           foreach (object o in col)
76              Enqueue (o);
77        }
78
79        public Queue (int initialCapacity, float growFactor)
80           requires initialCapacity >= 0 otherwise ArgumentOutOfRangeException;
81        {
82           //! if (initialCapacity < 0)
83           //!     throw new ArgumentOutOfRangeException("capacity", "Needs a non−negative
                 number");
84
85           //! Can't run this code, since Boogie currently doesn't know anything about floats
86           //! if (!(growFactor >= 1.0F && growFactor <= 10.0F))
87           //!     throw new ArgumentOutOfRangeException("growFactor", "Queue growth factor
                 must be between 1.0 and 10.0, inclusive");
88
89           _array = new object[initialCapacity];
90
91           //! Can't run this code, since Boogie currently doesn't know anything about floats
92           //! this._growFactor = (int)(growFactor * 100);
93           this._growFactor = 200;
94        }
95
96        // from ICollection
97
98        public virtual int Count {
99           [Pure]
100          get
101             ensures result >= 0;
102          {
```

```
103             return _size;
104          }
105       }
106
107       public virtual bool IsSynchronized {
108          [Pure]
109          get { return false; }
110       }
111
112       public virtual object! SyncRoot {
113          [Pure]
114          get
115             ensures result.IsPeerConsistent;
116          {
117             assume IsPeerConsistent; //! should be precondition
118             return this;
119          }
120       }
121
122       [Additive(false)]
123       public virtual void CopyTo (Array! array, int index)
124       {
125          //! if (array == null)
126          //!      throw new ArgumentNullException ("array");
127
128          if (index < 0)
129             throw new ArgumentOutOfRangeException ("index");
130
131          if (array.Rank > 1
132             || (index != 0 && index >= array.Length)
133             || _size > array.Length - index)
134             throw new ArgumentException ();
135
136          int contents_length = _array.Length;
137          int length_from_head = contents_length - _head;
138          // copy the _array of the circular array
139          //! Boogie can't handle Math.Min(), replace with equivalent code
140          //! Array.Copy (_array, _head, array, index,
141          //!        Math.Min (_size, length_from_head));
142          int len = (_size < length_from_head ? _size : length_from_head);
143          assume _array.IsPeerConsistent;
144          Array.Copy (_array, _head, array, index, len);
145
146          if (_size > length_from_head)
147             Array.Copy (_array, 0, array,
148                   index + length_from_head,
149                   _size - length_from_head);
150       }
151
152       // from IEnumerable
153
154       //! currently prevented by compiler bug
155       //! [Owned("peer")]
156       public /*virtual*/ IEnumerator! GetEnumerator () //! virtual causes CS0029 error
```

```
157         {
158             return new QueueEnumerator (this);
159         }
160
161         // from ICloneable
162
163                         //! currently prevented by compiler bug
164         [Additive(false)] //! [Owned("peer")]
165         public virtual object! Clone ()
166             ensures result is Queue;
167         {
168             Queue newQueue;
169
170             newQueue = new Queue (this._array.Length);
171             newQueue._growFactor = _growFactor;
172
173             //! This is established by the constructor
174             assume newQueue._array.Length == this._array.Length;
175             assume _array.IsPeerConsistent;
176             assume newQueue._array.IsPeerConsistent;
177             Array.Copy (this._array, 0, newQueue._array, 0,
178                   this._array.Length);
179             expose (newQueue) {
180                 newQueue._head = this._head;
181                 newQueue._size = this._size;
182                 newQueue._tail = this._tail;
183             }
184
185             return newQueue;
186         }
187
188         [Additive(false)]
189         public virtual void Clear ()
190             ensures _size == 0;
191         {
192             expose (this at Queue) {
193                 _version++;
194                 _head = 0;
195                 _size = 0;
196                 _tail = 0;
197                 for (int length = _array.Length - 1; length >= 0; length--)
198                     invariant length < _array.Length;
199                     invariant _head == _size && _head == _tail && _head == 0;
200                 {
201                     _array [length] = null;
202                 }
203                 //! Boogie can't infer this in/after loop
204                 assume _array.GetType() == typeof(object);
205             }
206         }
207
208         [Pure] [Additive(false)]
209         public virtual bool Contains (object obj)
210             requires obj == null || obj.IsPeerConsistent;
```

```
211         {
212            int tail = _head + _size;
213            if (obj == null) {
214               for (int i = _head; i < tail; i++)
215                  invariant i >= 0;
216               {
217                  //! valid assumption: if the _array.Length is 0,
218                  //! _head == tail, and the loop isn't executed.
219                  assume _array.Length > 0;
220                  if (_array[i % _array.Length] == null)
221                     return true;
222               }
223            } else {
224               for (int i = _head; i < tail; i++)
225                  invariant i >= 0;
226                  invariant obj.IsPeerConsistent;
227               {
228                  //! valid assumption: if the _array.Length is 0,
229                  //! _head == tail, and the loop isn't executed.
230                  assume _array.Length > 0;
231                  //! It's impossible to say whether Queue elements are
232                  //! peer consistent or not. We know nothing about their
233                  //! owner, we only hold read-only references to them!
234                  assume _array[i % _array.Length].IsPeerConsistent;
235                  if (obj.Equals (_array[i % _array.Length]))
236                     return true;
237               }
238            }
239            return false;
240         }
241
242         [Additive(false)]
243         public virtual object Dequeue ()
244            requires _size >= 1 otherwise InvalidOperationException;
245            ensures _size == old(_size) - 1;
246         {
247            expose (this at Queue) {
248               _version++;
249            }
250            //! if (_size < 1)
251            //!     throw new InvalidOperationException ();
252            object result = _array[_head];
253            expose (this at Queue) {
254               _array [_head] = null;
255               _head = (_head + 1) % _array.Length;
256               _size--;
257            }
258            assume result.IsPeerConsistent;
259            return result;
260         }
261
262         [Additive(false)]
263         public virtual void Enqueue (object obj)
264            ensures _size == old(_size) + 1;
```

```
265         {
266             expose (this at Queue) {
267                 _version++;
268             }
269             if (_size == _array.Length) {
270                 grow ();
271             }
272
273             expose (this at Queue) {
274                 _array[_tail] = obj;
275                 _tail = (_tail+1) % _array.Length;
276                 _size++;
277             }
278         }
279
280         [Pure] [Additive(false)]
281         public virtual object Peek ()
282             requires _size >= 1 otherwise InvalidOperationException;
283         {
284             //! if (_size < 1)
285             //!     throw new InvalidOperationException ();
286             assume _array[_head].IsPeerConsistent;
287             return _array[_head];
288         }
289
290         public static Queue Synchronized (Queue! queue) {
291             //! if (queue == null) {
292             //!     throw new ArgumentNullException ("queue");
293             //! }
294             return new SyncQueue (queue);
295         }
296
297         [Additive(false)]
298         public virtual object[]! ToArray ()
299         {
300             object[] ret = new object[_size];
301             CopyTo (ret, 0);
302             return ret;
303         }
304
305         [Additive(false)]
306         public virtual void TrimToSize()
307         {
308             expose (this at Queue) {
309                 _version++;
310             }
311             object[] trimmed = new object [_size];
312             CopyTo (trimmed, 0);
313             expose (this at Queue) {
314                 _array = trimmed;
315                 _head = 0;
316                 _tail = 0;
317             }
318         }
```

```
319
320         // private methods
321
322         private void grow ()
323            requires IsPeerConsistent;
324            ensures _array.Length > old(_array.Length);
325         {
326            int newCapacity = (_array.Length * _growFactor) / 100;
327            if (newCapacity < _array.Length + 1)
328               newCapacity = _array.Length + 1;
329            object[] newContents = new object[newCapacity];
330            CopyTo (newContents, 0);
331            expose (this at Queue) {
332               _array = newContents;
333               _head = 0;
334               _tail = _head + _size;
335            }
336         }
337
338         // private classes
339
340         private class SyncQueue : Queue {
341            [Owned("peer")] Queue queue;
342            invariant queue != null;
343
344            [Captured] [NotDelayed]
345            internal SyncQueue (Queue! queue)
346               requires queue.IsPeerConsistent;
347               ensures IsPeerConsistent;
348            {
349               Owner.AssignSame(this, queue);
350               this.queue = queue;
351            }
352
353            public override int Count {
354               [Pure] [Additive(false)]
355               get {
356                  lock (queue) {
357                     return queue.Count;
358                  }
359               }
360            }
361
362            public override bool IsSynchronized {
363               [Pure]
364               get {
365                  return true;
366               }
367            }
368
369            public override object! SyncRoot {
370               [Pure] [Additive(false)]
371               get {
372                  return queue.SyncRoot;
```

```
373             }
374          }
375
376          [Additive(false)]
377          public override void CopyTo (Array! array, int index) {
378             lock (queue) {
379                queue.CopyTo (array, index);
380             }
381          }
382
383 //!      public override IEnumerator! GetEnumerator () {
384 //!         assume queue.IsPeerConsistent;
385 //!         lock (queue) {
386 //!            return queue.GetEnumerator ();
387 //!         }
388 //!      }
389
390          [Additive(false)]
391          public override object! Clone () {
392             lock (queue) {
393                return new SyncQueue((Queue!) queue.Clone ());
394             }
395          }
396
397 /*
398          public override bool IsReadOnly {
399             get {
400                lock (queue) {
401                   return queue.IsReadOnly;
402                }
403             }
404          }
405 */
406
407          [Additive(false)]
408          public override void Clear () {
409             lock (queue) {
410                queue.Clear ();
411             }
412             assume queue._size == 0 ==> _size == 0;
413          }
414
415          [Additive(false)]
416          public override void TrimToSize () {
417             lock (queue) {
418                queue.TrimToSize ();
419             }
420          }
421
422          [Pure] [Additive(false)]
423          public override bool Contains (object obj) {
424             lock (queue) {
425                return queue.Contains (obj);
426             }
```

```
427         }
428
429         [Additive(false)]
430         public override object Dequeue () {
431            assume queue._size >= 1;
432            lock (queue) {
433               return queue.Dequeue ();
434            }
435         }
436
437         [Additive(false)]
438         public override void Enqueue (object obj) {
439            lock (queue) {
440               queue.Enqueue (obj);
441            }
442         }
443
444         [Pure] [Additive(false)]
445         public override object Peek () {
446            assume queue._size >= 1;
447            lock (queue) {
448               return queue.Peek ();
449            }
450         }
451
452         [Additive(false)]
453         public override object[]! ToArray () {
454            lock (queue) {
455               return queue.ToArray ();
456            }
457         }
458      }
459
460      [Serializable]
461      private class QueueEnumerator : IEnumerator, ICloneable {
462         [Owned("peer")] Queue queue;
463         private int _version;
464         private int current;
465
466         invariant queue != null;
467
468         [Captured] [NotDelayed]
469         internal QueueEnumerator (Queue! q)
470            requires q.IsPeerConsistent;
471            ensures IsPeerConsistent;
472         {
473            Owner.AssignSame(this, q);
474            queue = q;
475            _version = q._version;
476            current = -1; // one element before the _head
477         }
478
479         public object! Clone ()
480            requires IsPeerConsistent;
```

```
481          {
482             assume queue.IsPeerConsistent;
483             QueueEnumerator! q = new QueueEnumerator (queue);
484             q._version = _version;
485             q.current = current;
486             return q;
487          }
488
489          public virtual object Current {
490             get {
491                if (_version != queue._version
492                   || current < 0
493                   || current >= queue._size)
494                   throw new InvalidOperationException ();
495
496                //! Valid assumption: An exception is thrown above if queue._size is 0,
497                //! which implies that if we get here, the array Length is > 0, and
498                //! current is non−negative.
499                assume queue._array.Length > 0;
500                assume current >= 0;
501                //! original code
502                //! return queue._array[(queue._head + current) % queue._array.Length];
503                int i = (queue._head + current) % queue._array.Length;
504                assume i >= 0; //! queue._head is non−negative (invariant), as is current
505                assume i < queue._array.Length; //! obvious!
506                assume queue._array[i].IsPeerConsistent;
507                return queue._array[i];
508             }
509          }
510
511          public virtual bool MoveNext () {
512             if (_version != queue._version) {
513                throw new InvalidOperationException ();
514             }
515
516             if (current >= queue._size - 1) {
517                current = Int32.MaxValue; // to late!
518                return false;
519             } else {
520                current++;
521                return true;
522             }
523          }
524
525          public virtual void Reset () {
526             if (_version != queue._version) {
527                throw new InvalidOperationException();
528             }
529             current = -1;
530          }
531       }
532    }
533 }
```

A.3. Stack

```
1  //
2  // System.Collections.Stack
3  //
4  // Author:
5  //     Garrett Rooney (rooneg@electricjellyfish.net)
6  //
7  // (C) 2001 Garrett Rooney
8  //
9
10 //
11 // Copyright (C) 2004 Novell, Inc (http://www.novell.com)
12 //
13 // Permission is hereby granted, free of charge, to any person obtaining
14 // a copy of this software and associated documentation files (the
15 // "Software"), to deal in the Software without restriction, including
16 // without limitation the rights to use, copy, modify, merge, publish,
17 // distribute, sublicense, and/or sell copies of the Software, and to
18 // permit persons to whom the Software is furnished to do so, subject to
19 // the following conditions:
20 //
21 // The above copyright notice and this permission notice shall be
22 // included in all copies or substantial portions of the Software.
23 //
24 // THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
25 // EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
26 // MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
27 // NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
       BE
28 // LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
29 // OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
30 // WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
31 //
32
33 using System.Runtime.InteropService;
34 using Microsoft.Contracts;
35
36 namespace System.Collections {
37
38 #if NET_2_0
39    [ComVisible(true)]
40 #endif
41    [Serializable]
42    public class Stack : ICollection, IEnumerable, ICloneable {
43
44       // properties
45       private object[]! contents;
46       private int current = -1;
47       [SpecPublic] private int count;
48       private int capacity;
49       private int modCount;
50
```

```
51      const int default_capacity = 16;
52
53      invariant capacity == contents.Length;
54      invariant 0 <= count && count <= capacity;
55      invariant current == count - 1;
56
57      private void Resize(int ncapacity)
58         requires count <= ncapacity;
59         requires IsPeerConsistent;
60         ensures capacity >= ncapacity;
61         ensures count == old(count);
62         ensures forall { int i in (0:count - 1), contents[i] == old(contents[i])
               };
63      {
64         //! This was the original code here. However, it appears that
65         //! Math.Max() does not have any specifications, so I replaced
66         //! it with equivalent code that Boogie can handle.
67         //!ncapacity = Math.Max (ncapacity, 16);
68         if (ncapacity < 16) { ncapacity = 16; }
69
70         object[]! ncontents = new object[ncapacity];
71
72         assume contents.IsPeerConsistent;
73         Array.Copy(contents, ncontents, count);
74
75         expose (this at Stack) {
76            capacity = ncapacity;
77            contents = ncontents;
78         }
79      }
80
81      public Stack ()
82         ensures count == 0;
83      {
84         contents = new object[default_capacity];
85         capacity = default_capacity;
86      }
87
88      public Stack(ICollection col) : this (col == null ? 16 : col.Count)
89         requires col != null otherwise ArgumentNullException;
90         requires col.IsPeerConsistent;
91      {
92         //! if (col == null)
93         //!      throw new ArgumentNullException("col");
94
95                  // We have to do this because msft seems to call the
96                  // enumerator rather than CopyTo. This affects classes
97                  // like bitarray.
98         foreach (object o in col)
99            Push (o);
100     }
101
102     public Stack (int initialCapacity)
103        requires initialCapacity >= 0 otherwise ArgumentOutOfRangeException;
```

```
104         {
105             //! if (initialCapacity < 0)
106             //!     throw new ArgumentOutOfRangeException ("initialCapacity");
107
108             capacity = initialCapacity;
109             contents = new object[capacity];;
110         }
111
112         [Serializable]
113         private class SyncStack : Stack {
114
115             [Owned("peer")] Stack stack;
116             invariant stack != null;
117
118             [Captured] [NotDelayed]
119             internal SyncStack(Stack! s)
120                 ensures IsPeerConsistent;
121             {
122                 Owner.AssignSame(this, s);
123                 stack = s;
124             }
125
126             public override int Count {
127                 [Pure] [Additive(false)]
128                 get {
129                     lock (stack) {
130                         return stack.Count;
131                     }
132                 }
133             }
134
135 /*
136             public override bool IsReadOnly {
137                 get {
138                     lock (stack) {
139                         return stack.IsReadOnly;
140                     }
141                 }
142             }
143 */
144
145             public override bool IsSynchronized {
146                 [Pure]
147                 get { return true; }
148             }
149
150             public override object! SyncRoot {
151                 [Pure] [Additive(false)]
152                 get
153                 {
154                     return stack.SyncRoot;
155                 }
156             }
157
```

```
158         [Additive(false)]
159         public override void Clear() {
160            lock(stack) {
161               stack.Clear();
162            }
163            assume stack.count == 0 ==> count == 0;
164         }
165
166         [Additive(false)]
167         public override object! Clone() {
168            lock (stack) {
169               return Stack.Synchronized((Stack!)stack.Clone());
170            }
171         }
172
173         [Pure] [Additive(false)]
174         public override bool Contains(object obj) {
175            lock (stack) {
176               return stack.Contains(obj);
177            }
178         }
179
180         [Additive(false)]
181         public override void CopyTo(Array! array, int index) {
182            lock (stack) {
183               stack.CopyTo(array, index);
184            }
185         }
186
187 //!      public override IEnumerator! GetEnumerator() {
188 //!         lock (stack) {
189 //!            return new Enumerator(stack);
190 //!         }
191 //!      }
192
193         [Pure] [Additive(false)]
194         public override object Peek() {
195            //! required because using Count in a precondition doesn't work
196            assume stack.count > 0;
197            lock (stack) {
198               return stack.Peek();
199            }
200         }
201
202         [Additive(false)]
203         public override object Pop() {
204            //! required because using Count in a precondition doesn't work
205            assume stack.count > 0;
206            lock (stack) {
207               return stack.Pop();
208            }
209         }
210
211         [Additive(false)]
```

```
212             public override void Push(object obj) {
213                lock (stack) {
214                   stack.Push(obj);
215                }
216             }
217
218             [Additive(false)]
219             public override object[] ToArray() {
220                lock (stack) {
221                   return stack.ToArray();
222                }
223             }
224          }
225
226          public static Stack! Synchronized(Stack! s) {
227             //! if (s == null) {
228             //!     throw new ArgumentNullException();
229             //! }
230
231             return new SyncStack(s);
232          }
233
234          public virtual int Count
235          {
236             [Pure] get { return count; }
237          }
238
239    /*
240          public virtual bool IsReadOnly {
241             get { return false; }
242          }
243    */
244
245          public virtual bool IsSynchronized {
246             [Pure]
247             get { return false; }
248          }
249
250          public virtual object! SyncRoot {
251             [Pure] [Additive(false)]
252             get {
253                return this;
254             }
255          }
256
257          [Additive(false)]
258          public virtual void Clear()
259             ensures count == 0;
260          {
261             expose (this at Stack) {
262                modCount++;
263
264                for (int i = 0; i < count; i++)
265                   invariant count <= capacity;
```

```
266                invariant capacity == contents.Length;
267             {
268                contents[i] = null;
269             }
270
271             count = 0;
272             current = -1;
273          }
274       }
275
276       [Additive(false)]
277       public virtual object! Clone()
278          ensures result is Stack;
279       {
280          assume contents.IsPeerConsistent;
281          Stack stack = new Stack (contents);
282          //! should ideally be a postcondition of the constructor, but
283          //! that doesn't work yet.
284          assume stack.count == count;
285
286          //! The follwing code is not necessary because the count is
287          //! set correctly automatically through the constructor.
288          //! However it is the original Mono code.
289          expose (stack) {
290             stack.current = current;
291             stack.count = count;
292          }
293          return stack;
294       }
295
296       [Pure] [Additive(false)]
297       public virtual bool Contains(object obj)
298          requires obj != null ==> obj.IsPeerConsistent;
299       {
300          if (count == 0)
301             return false;
302
303          if (obj == null) {
304                for (int i = 0; i < count; i++) {
305                   if (contents[i] == null)
306                      return true;
307                }
308          } else {
309                for (int i = 0; i < count; i++) {
310                   //! guaranteed by invariant and precondition
311                   assume contents[i] != null ==> contents[i].IsPeerConsistent;
312                   if (obj.Equals (contents[i]))
313                      return true;
314                }
315          }
316
317          return false;
318       }
319
```

```
320
321         //! These would be the preconditions. They're not allowed here though
322         //! since this is an interface implementation. Consider them internal
323         //! documentation therefore:
324         //!
325         //!    requires index >= 0 otherwise ArgumentOutOfRangeException;
326         //!    requires array.Rank == 1 otherwise ArgumentException;
327         //!    requires array.Length == 0 || index < array.Length otherwise ArgumentException;
328         //!    requires count <= array.Length - index otherwise ArgumentException;
329         //!    requires array.IsPeerConsistent;
330         //!
331         [Additive(false)]
332         public virtual void CopyTo (Array! array, int index)
333         {
334            if (array == null) {
335               throw new ArgumentNullException("array");
336            }
337
338            if (index < 0) {
339               throw new ArgumentOutOfRangeException("index");
340            }
341
342            if (array.Rank > 1 ||
343               array.Length > 0 && index >= array.Length ||
344               count > array.Length - index) {
345               throw new ArgumentException();
346            }
347
348            for (int i = current; i != -1; i--)
349               invariant i <= current && i >= -1;
350               invariant array.IsPeerConsistent;
351               invariant array.Length >= count + index;
352            {
353               assert i >= 0;
354               //! precondition and invariant
355               assume contents[i] != null ==> contents[i].IsPeerConsistent;
356               array.SetValue(contents[i], count - (i + 1) + index);
357            }
358         }
359
360         private class Enumerator : IEnumerator, ICloneable {
361
362            const int EOF = -1;
363            const int BOF = -2;
364
365            [Owned("peer")] Stack stack;
366            private int modCount;
367            private int current;
368
369            invariant stack != null;
370            invariant current >= -2;
371
372            [Captured] [NotDelayed]
373            internal Enumerator(Stack! s)
```

```
374             ensures IsPeerConsistent;
375         {
376             Owner.AssignSame(this, s);
377             stack = s;
378             modCount = s.modCount;
379             current = BOF;
380         }
381
382         public object! Clone ()
383         {
384             return MemberwiseClone ();
385         }
386
387         public virtual object Current {
388             get {
389                 if (modCount != stack.modCount
390                     || current == BOF
391                     || current == EOF
392                     || current > stack.count)
393                     throw new InvalidOperationException();
394                 assert current >= 0;
395                 //! preceding if
396                 assume current >= 0 && current <= stack.count;
397                 //! current starts at stack.current, then is only decremented
398                 //! stack.current stays constant while modCount == stack.modCount
399                 assume current <= stack.current;
400                 assume stack.current == stack.count - 1; //! stack invariant
401                 assume stack.count <= stack.contents.Length; //! stack invariant
402                 assume stack.contents[current] != null ==> stack.contents[current].
                        IsPeerConsistent;
403                 return stack.contents[current];
404             }
405         }
406
407         public virtual bool MoveNext() {
408             if (modCount != stack.modCount)
409                 throw new InvalidOperationException();
410
411             switch (current) {
412             case BOF:
413                 expose (this) {
414                     //! Stack invariant
415                     assume stack.current >= -1 && stack.current < stack.count;
416                     current = stack.current;
417                 }
418                 return current != -1;
419
420             case EOF:
421                 return false;
422
423             default:
424                 expose (this) {
425                     current--;
426                 }
```

```
427                return current != -1;
428             }
429          }
430
431          public virtual void Reset() {
432             if (modCount != stack.modCount) {
433                throw new InvalidOperationException();
434             }
435
436             expose (this) {
437                current = BOF;
438             }
439          }
440       }
441
442       //! currently prevented by compiler bug
443       //! [Owned("peer")]
444       public /*virtual*/ IEnumerator! GetEnumerator() { //! virtual causes CS0029 error
445          return new Enumerator(this);
446       }
447
448       [Pure] [Additive(false)]
449       public virtual object Peek()
450          requires count > 0 otherwise InvalidOperationException;
451          requires IsPeerConsistent;
452       {
453          if (current == -1) {
454             throw new InvalidOperationException();
455          } else {
456             assume contents[current] != null ==> contents[current].IsPeerConsistent
                    ;
457             return contents[current];
458          }
459       }
460
461       [Additive(false)]
462       public virtual object Pop()
463          requires count > 0;
464          requires IsPeerConsistent;
465          ensures count == old(count) - 1;
466          //ensures result == old(Peek());
467       {
468          if (current == -1) {
469             throw new InvalidOperationException();
470          } else {
471             object ret;
472
473             expose (this at Stack) {
474                modCount++;
475
476                ret = contents[current];
477                contents [current] = null;
478
479                count--;
```

```
480             current--;
481          }
482          // if we're down to capacity/4, go back to a
483          // lower array size.  this should keep us from
484          // sucking down huge amounts of memory when
485          // putting large numbers of items in the Stack.
486          // if we're lower than 16, don't bother, since
487          // it will be more trouble than it's worth.
488          if (count <= (capacity/4) && count > 16) {
489             Resize(capacity/2);
490          }
491
492          //! invariant and precondition
493          assume ret != null ==> ret.IsPeerConsistent;
494          return ret;
495       }
496    }
497
498    [Additive(false)]
499    public virtual void Push(Object o)
500       requires IsPeerConsistent;
501       requires o == null || o.IsPeerConsistent;
502       ensures count == old(count) + 1;
503       ensures Peek() == o;
504    {
505       modCount++;
506
507       if (capacity == count) {
508          Resize(capacity * 2);
509          assert capacity >= count * 2;
510       }
511       else { //! else block added for illustration
512          assert capacity > count;
513       }
514       //! even though both asserts above hold, Boogie doesn't
515       //! infer this property here.
516       assume capacity > count;
517
518       expose (this at Stack) {
519          count++;
520          current++;
521
522          assume contents.IsPeerConsistent; //! invariant and precondition
523          assume o != null ==> o.IsPeerConsistent; //! precondition
524          contents.SetValue(o, current);
525       }
526
527       //! because o and Peek() both are contents[current]
528       assume o == Peek();
529    }
530
531    public virtual object[] ToArray()
532       requires IsPeerConsistent;
533    {
```

```
534             object[] ret = new object[count];
535
536             assume contents.IsPeerConsistent; //! precondition
537             Array.Copy(contents, ret, count);
538
539             // ret needs to be in LIFO order
540             Array.Reverse(ret);
541
542             return ret;
543         }
544     }
545 }
```

A.4. SimpleStack, C# Version

```
1  // SimpleStack, based on Mono's System.Collections.Stack,
2  // the copyright notice of which is below.
3  //
4  // System.Collections.Stack
5  //
6  // Author:
7  //     Garrett Rooney (rooneg@electricjellyfish.net)
8  //
9  // (C) 2001 Garrett Rooney
10 //
11 // Copyright (C) 2004 Novell, Inc (http://www.novell.com)
12 //
13 // Permission is hereby granted, free of charge, to any person obtaining
14 // a copy of this software and associated documentation files (the
15 // "Software"), to deal in the Software without restriction, including
16 // without limitation the rights to use, copy, modify, merge, publish,
17 // distribute, sublicense, and/or sell copies of the Software, and to
18 // permit persons to whom the Software is furnished to do so, subject to
19 // the following conditions:
20 //
21 // The above copyright notice and this permission notice shall be
22 // included in all copies or substantial portions of the Software.
23 //
24 // THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
25 // EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
26 // MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
27 // NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
       BE
28 // LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
29 // OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
30 // WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
31 //
32
33 using System;
34 using System.Collections;
35
36 public class SimpleStack : ICollection, IEnumerable {
37
38    // properties
39    private object[] contents;
40    private int current = -1;
41    private int count;
42    private int capacity;
43    private int modCount;
44
45    const int default_capacity = 16;
46
47    private void Resize(int ncapacity)
48    {
49       ncapacity = Math.Max (ncapacity, 16);
50       object[] ncontents = new object[ncapacity];
```

```
51
52          Array.Copy(contents, ncontents, count);
53
54          capacity = ncapacity;
55          contents = ncontents;
56      }
57
58      public SimpleStack ()
59      {
60          contents = new object[default_capacity];
61          capacity = default_capacity;
62      }
63
64      public virtual int Count {
65          get { return count; }
66      }
67
68      public virtual bool IsSynchronized {
69          get { return false; }
70      }
71
72      public virtual object SyncRoot {
73          get { return this; }
74      }
75
76      public virtual void Clear() {
77          modCount++;
78
79          for (int i = 0; i < count; i++) {
80              contents[i] = null;
81          }
82
83          count = 0;
84          current = -1;
85      }
86
87      public virtual void CopyTo (Array array, int index) {
88          if (array == null) {
89              throw new ArgumentNullException("array");
90          }
91
92          if (index < 0) {
93              throw new ArgumentOutOfRangeException("index");
94          }
95
96          if (array.Rank > 1 ||
97              array.Length > 0 && index >= array.Length ||
98              count > array.Length - index) {
99              throw new ArgumentException();
100         }
101
102         for (int i = current; i != -1; i--) {
103             array.SetValue(contents[i],
104                     count - (i + 1) + index);
```

```
105         }
106      }
107
108      private class Enumerator : IEnumerator {
109
110         const int EOF = -1;
111         const int BOF = -2;
112
113         SimpleStack stack;
114         private int modCount;
115         private int current;
116
117         internal Enumerator(SimpleStack s) {
118            stack = s;
119            modCount = s.modCount;
120            current = BOF;
121         }
122
123         public virtual object Current {
124            get {
125               if (modCount != stack.modCount
126                  || current == BOF
127                  || current == EOF
128                  || current > stack.count)
129                  throw new InvalidOperationException();
130               return stack.contents[current];
131            }
132         }
133
134         public virtual bool MoveNext() {
135            if (modCount != stack.modCount)
136               throw new InvalidOperationException();
137
138            switch (current) {
139            case BOF:
140               current = stack.current;
141               return current != -1;
142
143            case EOF:
144               return false;
145
146            default:
147               current--;
148               return current != -1;
149            }
150         }
151
152         public virtual void Reset() {
153            if (modCount != stack.modCount) {
154               throw new InvalidOperationException();
155            }
156
157            current = BOF;
158         }
```

```
159     }
160
161     public virtual IEnumerator GetEnumerator() {
162        return new Enumerator(this);
163     }
164
165     public virtual object Peek() {
166        if (current == -1) {
167           throw new InvalidOperationException();
168        } else {
169           return contents[current];
170        }
171     }
172
173     public virtual object Pop() {
174        if (current == -1) {
175           throw new InvalidOperationException();
176        } else {
177           modCount++;
178
179           object ret = contents[current];
180           contents [current] = null;
181
182           count--;
183           current--;
184
185           // if we're down to capacity/4, go back to a
186           // lower array size.  this should keep us from
187           // sucking down huge amounts of memory when
188           // putting large numbers of items in the Stack.
189           // if we're lower than 16, don't bother, since
190           // it will be more trouble than it's worth.
191           if (count <= (capacity/4) && count > 16) {
192              Resize(capacity/2);
193           }
194
195           return ret;
196        }
197     }
198
199     public virtual void Push(Object o) {
200        modCount++;
201
202        if (capacity == count) {
203           Resize(capacity * 2);
204        }
205
206        count++;
207        current++;
208
209        contents[current] = o;
210     }
211  }
```

A.5. SimpleStack, Spec# Version

```
1  // SimpleStack, based on Mono's System.Collections.Stack,
2  // the copyright notice of which is below.
3  //
4  // System.Collections.Stack
5  //
6  // Author:
7  //     Garrett Rooney (rooneg@electricjellyfish.net)
8  //
9  // (C) 2001 Garrett Rooney
10 //
11 // Copyright (C) 2004 Novell, Inc (http://www.novell.com)
12 //
13 // Permission is hereby granted, free of charge, to any person obtaining
14 // a copy of this software and associated documentation files (the
15 // "Software"), to deal in the Software without restriction, including
16 // without limitation the rights to use, copy, modify, merge, publish,
17 // distribute, sublicense, and/or sell copies of the Software, and to
18 // permit persons to whom the Software is furnished to do so, subject to
19 // the following conditions:
20 //
21 // The above copyright notice and this permission notice shall be
22 // included in all copies or substantial portions of the Software.
23 //
24 // THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
25 // EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
26 // MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
27 // NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
       BE
28 // LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
29 // OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
30 // WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
31 //
32
33 using System;
34 using System.Collections;
35 using Microsoft.Contracts;
36
37 public class SimpleStack : ICollection, IEnumerable {
38
39     // properties
40     private object[]! contents;
41     private int current = -1;
42     [SpecPublic] private int count;
43     private int capacity;
44     private int modCount;
45
46     const int default_capacity = 16;
47
48     invariant count >= 0 && count <= contents.Length;
49     invariant current >= -1;
50     invariant current + 1 == count;
```

```
51    invariant contents.GetType() == typeof(object[]);
52
53    [Additive(false)]
54    private void Resize(int ncapacity)
55       requires ncapacity >= count;
56       requires contents.IsPeerConsistent;
57       ensures count == old(count);
58       ensures forall { int i in (0:count - 1), contents[i] == old(contents[i]) };
59    {
60       ncapacity = ncapacity < 16 ? 16 : ncapacity;
61       object[]! ncontents = new object[ncapacity];
62
63       Array.Copy(contents, ncontents, count);
64
65       capacity = ncapacity;
66       expose (this at SimpleStack) { contents = ncontents; }
67    }
68
69    public SimpleStack ()
70    {
71       contents = new object[default_capacity];
72       capacity = default_capacity;
73    }
74
75    public virtual int Count {
76       [Pure] get
77          ensures result >= 0;
78          ensures result == count;
79       {
80          return count;
81       }
82    }
83
84    public virtual bool IsSynchronized {
85       [Pure] get { return false; }
86    }
87
88    public virtual object! SyncRoot {
89       [Pure] get
90          ensures result.IsPeerConsistent;
91       {
92          assume IsPeerConsistent;
93          return this;
94       }
95    }
96
97    [Additive(false)]
98    public virtual void Clear()
99       ensures count == 0;
100   {
101      modCount++;
102
103      for (int i = 0; i < count; i++) {
104         contents[i] = null;
```

```
105         }
106
107         expose(this at SimpleStack) {
108             count = 0;
109             current = -1;
110         }
111     }
112
113     public virtual void CopyTo (Array! array, int index) {
114         if (index < 0) {
115             throw new ArgumentOutOfRangeException("index");
116         }
117
118         if (array.Rank > 1 ||
119             array.Length > 0 && index >= array.Length ||
120             count > array.Length - index) {
121             throw new ArgumentException();
122         }
123
124         for (int i = current; i != -1; i--)
125             invariant i >= -1 && i <= current;
126             invariant count <= array.Length - index;
127         {
128             assume contents[i] != null ==> contents[i].IsPeerConsistent;
129             array.SetValue(contents[i],
130                     count - (i + 1) + index);
131         }
132     }
133
134     private class Enumerator : IEnumerator {
135
136         const int EOF = -1;
137         const int BOF = -2;
138
139         SimpleStack! stack;
140         private int modCount;
141         private int current;
142
143         invariant current >= -2;
144
145         internal Enumerator(SimpleStack! s) {
146             stack = s;
147             modCount = s.modCount;
148             current = BOF;
149         }
150
151         public virtual object Current {
152             get {
153                 if (modCount != stack.modCount
154                     || current == BOF
155                     || current == EOF
156                     || current > stack.count)
157                     throw new InvalidOperationException();
158                 assert current >= -1 && current != -1 ==> current >= 0;
```

```
159             assume current < stack.contents.Length;
160             assume stack.contents[current] != null ==>
161                stack.contents[current].IsPeerConsistent;
162             return stack.contents[current];
163          }
164       }
165
166       [Additive(false)]
167       public virtual bool MoveNext() {
168          if (modCount != stack.modCount)
169             throw new InvalidOperationException();
170
171          switch (current) {
172          case BOF:
173             expose (this at Enumerator) {
174                assume stack.count <= stack.contents.Length;
175                assume stack.current >= -1;
176                assume stack.current + 1 == stack.count;
177                current = stack.current;
178             }
179             return current != -1;
180
181          case EOF:
182             return false;
183
184          default:
185             expose (this at Enumerator) {
186                current--;
187             }
188             return current != -1;
189          }
190       }
191
192       [Additive(false)]
193       public virtual void Reset() {
194          if (modCount != stack.modCount) {
195             throw new InvalidOperationException();
196          }
197
198          expose (this at Enumerator) {
199             current = BOF;
200          }
201       }
202    }
203
204    public /*virtual*/ IEnumerator! GetEnumerator() {
205       return new Enumerator(this);
206    }
207
208    [Pure]
209    public virtual object Peek()
210       requires count > 0 otherwise InvalidOperationException;
211    {
212       assume contents[current] != null ==> contents[current].IsPeerConsistent;
```

```
213         return contents[current];
214     }
215
216     [Additive(false)]
217     public virtual object Pop()
218         requires count > 0 otherwise InvalidOperationException;
219         //ensures result == old(Peek());
220         ensures count == old(count) - 1;
221     {
222         modCount++;
223
224         object ret = contents[current];
225         contents [current] = null;
226
227         expose (this at SimpleStack) {
228             count--;
229             current--;
230         }
231
232         // if we're down to capacity/4, go back to a
233         // lower array size.  this should keep us from
234         // sucking down huge amounts of memory when
235         // putting large numbers of items in the Stack.
236         // if we're lower than 16, don't bother, since
237         // it will be more trouble than it's worth.
238         if (count <= (capacity/4) && count > 16) {
239             assume contents.IsPeerConsistent;
240             Resize(capacity/2);
241         }
242
243         assume ret != null ==> ret.IsPeerConsistent;
244         return ret;
245     }
246
247     [Additive(false)]
248     public virtual void Push(Object o)
249         //ensures o == Peek();
250         ensures count == old(count) + 1;
251     {
252         modCount++;
253
254         if (capacity == count) {
255             assume contents.IsPeerConsistent;
256             Resize(capacity * 2);
257         }
258
259         assume count < contents.Length;
260
261         expose (this at SimpleStack) {
262             count++;
263             current++;
264         }
265
266         contents[current] = o;
```

```
267         }
268     }
```

B. Bibliography

[1] Microsoft Research. *Spec#*. URL `http://research.microsoft.com/specsharp/`.

[2] The Mono Project. *Mono*. URL `http://www.mono-project.com/`.

[3] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. *The Spec# Programming System: An Overview.* In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pp. 49–69. Springer, January 2005. ISBN 978-3-540-24287-1. ISSN 0302-9743. doi:10.1007/b105030. URL `http://research.microsoft.com/specsharp/papers/krml136.pdf`.

[4] Mike Barnett, Rob 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, June 2004. URL `http://www.jot.fm/issues/issue_2004_06/article2`.

[5] K. Rustan M. Leino and Peter Müller. *Modular Verification of Static Class Invariants*. In *FM 2005: Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pp. 26–42. Springer, July 2005. ISBN 978-3-540-27882-5. ISSN 0302-9743. doi:10.1007/b27882. URL `http://research.microsoft.com/~leino/papers/krml153.pdf`.

[6] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997. ISBN 0-13-629155-4.

[7] Microsoft Research. *Spec# 1.0.6404 Release Notes*. URL `http://research.microsoft.com/specsharp/1.0.6404/relnotes.htm`.

[8] Compaq Systems Research Center. *ESC/Java*. URL `http://research.compaq.com/SRC/esc/`.