Implementation of Frozen Objects into Spec#

Florian Leu

Master Thesis Report

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

http://www.pm.inf.ethz.ch/

September 2009

Supervised by:
Joseph N. Ruskiewicz
Prof. Dr. Peter Müller

Chair of Programming Methodology
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

The Spec# programming system allows declaring immutable classes to support the familiar concept of object immutability. However, this class-based approach is restrictive, because it prevents making instances of arbitrary classes immutable or to decide when an object should become immutable. This thesis describes the implementation of the frozen objects methodology, which allows overcoming the aforementioned restrictions, into Spec# and shows how to replace the previous implementation for immutable classes with one depending on frozen objects.

Contents

Chapter 1

Introduction

Immutability of objects is an extremely useful concept, because it allows other objects to depend on the fact that the state of such objects is never modified. In particular, this means that other objects may have invariants depending on an immutable object without owning it, allowing multiple objects to have invariants depending on the same object. Especially for static program verification, it is important to have strong contracts, enabling a verifier to prove certain conditions. The guaranteed immutability of an object is therefore of great use, because it gives the strongest possible contract for an object's state, namely that it is never altered.

The Spec# programming system supports the concept of immutable types by means of immutable classes. A class can be declared as immutable by prefixing an [Immutable] modifier, as seen here for class *Foo*:

```
[Immutable]
public class Foo {
  ...
}
```

An object that is of an immutable type may only be initialized by its constructor, but is immutable as soon as the constructor has terminated. This concept is helpful in many practical applications, particularly because it allows mentioning objects of an immutable type in other objects' invariants, but there are also some severe drawbacks and limitations. With immutable classes, it is for example not possible to have mutable as well as immutable instances of one particular type.

In the following, we take a look at and address those limitations of immutable classes by implementing the frozen objects methodology, originally described by Leino, Müller, and Wallenburg [7], into Spec#. We also extend the static program verifier Boogie, which is able to verify Spec# programs, to support the newly implemented frozen objects. This technique allows us to declare immutability on an object level instead of a class level. Amongst other advantages discussed later, frozen objects help to overcome the above mentioned problem of not being able to have mutable as well as immutable instances of one class.

We still keep the concept of immutable classes for reasons of backward compatibility, but also because if one needs only immutable objects of a specific class, it is more convenient to just declare this directly on the class level instead of separately on each individual object.

However, we replace the previous implementation of immutable classes with one that makes use of the frozen objects methodology. Doing so guarantees that the concept of immutability is not implemented in two different ways afterwards. It also allows us to remove many special case distinctions from the source code of Boogie, which have been necessary to distinguish between the handling of objects of a mutable or an immutable type.

The frozen objects technique, on the other hand, relies mainly on already existing checks from Spec#'s ownership model, which not only simplifies the source code of Boogie, but also reduces the size of files compiled for static verification into the BoogiePL language.

The remainder of this thesis is structured as follows. Chapter 2 provides all the necessary background information about Spec#, Boogie, immutable classes, and frozen objects. Chapter 3 discusses the implementation of the frozen objects methodology into Spec#, Chapter 4 the encoding of the frozen objects theory, and Chapter 5 the necessary extensions in Boogie, enabling us to verify programs using frozen objects. In Chapter 6, we show how the concept of immutable classes is implemented with frozen objects, and discuss why strings are a special case and have to be treated differently. In Chapter 7, we conclude this thesis by talking about the testing of the implementation and pointing out the limitations of frozen objects.

Chapter 2

Background

2.1 Spec#

The Spec# programming language [1] [8] is an extension of C#, adding features to the language that help verifying the correctness of a program. We discuss here in detail only those features of Spec# relevant for the implementation of the frozen objects methodology.

Spec# adds non-null types, method pre- and postconditions, as well as object and loop invariants to C#. Pre- and postconditions need to hold on entry and exit of a method respectively. The caller of a method is obligated to satisfy the preconditions and can assume the postconditions to hold at the end of the method call.

Spec# includes an ownership model [2], where each object can be owned by at most one other object. An owner consists of an (object, class frame) pair, where the class frame is a type of the object. Ownership transfer is not allowed in Spec#, meaning that once an object has been assigned an owner, this owner cannot be changed anymore.

A field of an object may be declared as either [Peer], meaning that the field has the same owner as the object in which it is declared, or [Rep], meaning that the field is owned by the object in which it is declared. Thus, the [Peer] and [Rep] modifiers structure the object instances into an ownership graph with sibling (peer) and child (rep) nodes.

With the modifiers [ElementsPeer] and [ElementsRep], the elements of an array can be declared as either peer or rep. Also, the type parameters of a generic class may be marked with those two modifiers. For example, if we have a generic list field in an object o with its type parameter specified as [ElementsPeer], then all fields typed with this parameter in the list class are peers of object o.

Object fields mentioned in an invariant have to be exposed before they are allowed to be modified. This can be achieved either with the *expose* keyword of Spec#, which locally exposes the object for the current class frame only, or with *additively expose*, which requires the object to already be additively exposed for all subclass frames and guarantees that the object is exposed for the current and all subclass frames. Invariants of an object may temporarily be broken as long as this object is exposed, but they need to hold again at the end of the expose block.

In order to modify the state of an owned object, it is required that the owner object is locally or additively exposed for the class frame given in the (object, class frame) pair denoting the owner. This implies that an assignment to a [Rep] field requires *this* to be exposed.

On entry of a method, the parameters and the receiver of the method are by default required to be *peer consistent*, meaning that the object and all of its peers have to be *consistent*. An object is consistent, if it has been fully constructed, is not exposed, and has either no owner or the owner is exposed. This implies that if an object is peer consistent, its invariants are guaranteed to hold, because it cannot be peer consistent and exposed at the same time.

A method is *pure*, if it does not alter the state of any object. The receiver and the parameters of a pure method do not need to be peer consistent but only *peer valid*, where *valid* is defined as an

object being fully constructed and not exposed. In contrast to being consistent, being valid does not say anything about the owner, the reason being that neither the receiver's nor a parameter's state gets modified inside a pure method. Hence, it does not matter whether the owner is exposed.

If an object is not valid, then it is *mutable*, allowing its invariants to be broken. A mutable object can either be *locally mutable*, meaning that the object has been exposed only for its current class frame, or *additively mutable*, meaning that the object has been additively exposed for its current class frame and all subclass frames. Being valid or mutable for an object of class T can also be encoded with two variables *inv* and *localinv* [5]:

valid	if o.inv $<:$ T $\wedge$ o.localinv $\neq$ base(T)
additively mutable	if $\neg$(o.inv $<:$ T)
locally mutable	if o.localinv = base(T) $\vee$ $\neg$(o.inv $<:$ T)

where base(T) is the immediate superclass of T and $<:$ denotes a subtype relationship. All subclasses of the class the variable *inv* is set to are additively exposed, while *localinv* is set to the base class of the class frame that is currently locally exposed.

The *frame condition* of a method states, which objects and fields are allowed to be modified by the method. The fields of the *this* instance are the only ones that by default are included in the frame condition. If other objects are to be modified inside a method, then those objects have to be added to the *modifies* clause of this method. However, objects owned by the object containing a method do not have to be included in its modifies clause. Methods are also not allowed to call other methods with less restrictive frame conditions.

The Spec# compiler takes a Spec# source code file and emits .NET assemblies either as process assemblies (.exe) or library assemblies (.dll). In addition, the compiler also generates runtime checks for method contracts, object invariants, and loop invariants. For performance reasons, not more checks are done during runtime.

Checking for the correct use of the ownership model or verifying that the frame condition of a method is never violated is the task of static program verification. We introduce one such static program verifier, with which we may verify Spec# programs, in the next section.

2.2 Boogie

Boogie [3] is a static program verifier that takes a BoogiePL program (.bpl) and generates verification conditions which are then passed to an SMT solver (by default the Z3 SMT solver [6]). Boogie can also take a compiled Spec# program (.exe or .dll) as input and first translate it to a BoogiePL program, before generating the verification conditions.

Boogie verifies a program method by method, i.e. it follows a modular approach. For that to work, it relies heavily on strong method contracts. If a method is called inside another method, Boogie removes the method call and just asserts the preconditions of the method and assumes the postconditions of it. By doing so, the dependency between caller and callee is removed, enabling modularity of the methods.

A verification failure is returned, when Boogie cannot verify a program. This either indicates an error in the program code, or that too few information is available to prove a certain verification condition, indicating that the contracts are not strong enough.

2.2.1 BoogiePL

BoogiePL [4] (or Boogie 2 as the new version of the language is called) is an intermediate verification language used not to run but to statically verify a program. A compiled Spec# program is translated into BoogiePL by applying some transformations. We briefly discuss those parts of BoogiePL that are relevant to understand this thesis.

Type

A type is defined in BoogiePL with the keyword *type*, followed by the type name.

This example defines a reference type:

```
type ref;
```

Constant

A constant in BoogiePL is defined with the keyword *const*, followed by the name of the constant and its type.

This example defines the *null* constant:

```
const null : ref;
```

Function

A function is defined in BoogiePL with the keyword *function*, followed by the function name, its parameter types, and the return value type. There is no actual implementation of the function, which is why it is also called an *uninterpreted function*. The function gets its intended meaning through axioms as discussed below.

This example defines a function that determines for a given type whether it is a value type:

```
function $IsValueType(TName) returns (bool);
```

Axiom

An axiom in BoogiePL is defined with the keyword *axiom*, followed by an expression that evaluates to a boolean value.

This example defines an axiom saying that the type *System.Boolean* is a value type, thereby giving meaning to the function above:

```
axiom $IsValueType(System.Boolean);
```

Procedure

A procedure is defined in BoogiePL with the keyword *procedure*, followed by the procedure name and its parameter names and types. In addition, the procedure is described by contracts, i.e. modifies, requires, and ensures clauses. A procedure can optionally also have an implementation.

This example shows a procedure *Bar* with two parameters, the first being the receiver object. In BoogiePL, the implicit receiver gets always turned into an explicit parameter. The procedure requires *this* to be not *null*, and ensures *o* not to be *null* on return:

```
procedure Bar(this: ref, o: ref);
  requires this ≠ null;
  ensures o ≠ null;

implementation Bar(this: ref, o: ref)
{ ... }
```

Local Variable

A local variable is declared inside a method implementation in BoogiePL. With a where-clause, additional properties of the variable may be described.

In this example, a local variable is declared with the additional properties that it is of type *Foo* and allocated:

```
var local0: ref where $Is(local0, Foo) ∧ $Heap[local0, $allocated];
```

Generics

The generic parameter *alpha* is used in BoogiePL to allow a function or an axiom to be instantiated with different types.

The function *IsStaticField*, in this example, takes a field reference as input and determines whether it is a static field:

```
function IsStaticField(f: Field ref) returns (bool);
```

But if we also want to know whether integer or boolean fields are static, we would have to add another two functions similar to the one above:

```
function IsStaticField(f: Field int) returns (bool);
function IsStaticField(f: Field bool) returns (bool);
```

A better solution is to use the generic type parameter *alpha*, allowing all three functions to be combined into one, which can then be called with fields of all different types as input parameter:

```
function IsStaticField<alpha>(f: Field alpha) returns (bool);
```

Trigger

A trigger is used to tell an SMT solver how to instantiate universal quantifiers, which is crucial for getting desired results with a good performance. Usually, a prover has to instantiate a quantified variable with each possible instance. A trigger reduces this workload by telling a prover, such as Z3, which instances to pick. In BoogiePL, triggers can be added in curly braces after a universal quantifier and have to mention all the bound variables of it.

In this example, the trigger g(t) tells the prover to only select those instances of type T that occur as terms g(t) in the current proof context:

```
axiom (∀ t: T • { g(t) } f(g(t)) ≥ 4);
```

2.2.2 Prelude

Boogie is a program verifier that is independent from Spec# and, therefore, initially knows nothing about the rules of this language. *Prelude* is a file that provides Boogie with background theory of the Spec# language. In this file, amongst other things, the type system, the base types, the handling of arrays, and the ownership model are described in BoogiePL.

When the Boogie verifier is invoked for a compiled Spec# program, it not only translates the given assembly file into a BoogiePL file, but also prefixes the content of *Prelude* to it. Because only by having the background theory of the language available, is a prover then able to prove the verification conditions generated by Boogie for this program.

In what follows, we are discussing those parts of the *Prelude* file that are important to understand the addition of the frozen objects theory to it as discussed in Chapter 4.

Heap

The heap of a Spec# program, which is the part of memory where dynamically allocated objects are stored, is modeled in BoogiePL as a global array, mapping a given field of an object to a value.

In this example, the value 4 is stored in the field *intField* of the *Foo* object referenced by *stack0o*:

```
$Heap[stack0o, Foo.intField] := 4;
```

Ownership

Ownership of an object is encoded with the two constants *$ownerRef* and *$ownerFrame*, which represent the owner object and the owner class frame of an (object, class frame) owner tuple respectively:

```
const unique $ownerRef : Field ref;
const unique $ownerFrame : Field TName;
```

Whenever an object is yet unowned, its *$ownerFrame* field is set to the special *$PeerGroupPlaceholder* type that, as the name suggests, is a placeholder for the nonexistent owner and cannot be used in any other way:

```
const unique $PeerGroupPlaceholder : TName;
```

No owned object can have *$PeerGroupPlaceholder* as its class frame. This placeholder type is used mainly in axioms, assertions, pre-, and postconditions to distinguish between unowned and owned objects by checking whether *$ownerFrame* is equal to *$PeerGroupPlaceholder*.

Object Consistency and Validity

Whether an object is valid or mutable is encoded with the two constants *$inv* and *$localinv*:

```
const unique $inv : Field TName;
const unique $localinv : Field TName;
```

The constant *$inv* contains the name of the most derived class for which an object invariant holds, whereas *$localinv* holds the immediate supertype of the class for which an object invariant is allowed to be broken. Hence, in order for an object to be valid, *$localinv* may not be set to the immediate supertype of the object's type and *$inv* not to any supertype of it. The actual implementation guarantees even more by saying that if an object is valid, *$inv* and *$localinv* are both set to the exact type of the object:

```
$Heap[o, $inv] == $typeof(o) ∧ $Heap[o, $localinv] == $typeof(o)
```

Peer validity has to guarantee the above not only for the object itself but also for all of its peers, i.e. for all allocated objects with the same owner:

```
(∀ p: ref • p ≠ null ∧ $Heap[p, $allocated] ∧ $Heap[p, $ownerRef] ==
    $Heap[o, $ownerRef] ∧ $Heap[p, $ownerFrame] == $Heap[o, $ownerFrame] ⇒
    $Heap[p, $inv] == $typeof(p) ∧ $Heap[p, $localinv] == $typeof(p))
```

Peer consistency can be expressed as an object being peer valid and having either no owner or an exposed owner. Thus, we not only have to ensure that the above expression for peer validity holds, but also that the following expression about the owner of the object is satisfied:

```
$Heap[o, $ownerFrame] == $PeerGroupPlaceholder ∨
    ¬($Heap[$Heap[o, $ownerRef], $inv] <: $Heap[o, $ownerFrame]) ∨
    $Heap[$Heap[o, $ownerRef], $localinv] == $BaseClass($Heap[o, $ownerFrame])
```

The first part of this expression checks whether the object is unowned. In the second part, we check if the $*inv* field of $*ownerRef* does not contain a subtype of the type stored in $*ownerFrame*, because that would make the owner additively exposed. The last part then checks whether $*localinv* of $*ownerRef* is equal to $*ownerFrame*, as else the owner would be locally exposed.

2.3 Immutable Classes

2.3.1 General Concept

In Spec#, a class may be specified with an [Immutable] modifier. Objects that have a type corresponding to such an *immutable class* are guaranteed to be immutable after their initialization. Thus, after the constructor of such an object has terminated, the state of this object cannot be altered anymore for the rest of the object's lifecycle. Objects of an immutable type are therefore not allowed to be exposed, as that would allow them to be modified.

```
public class WeatherReport {
  private TempMeasurement temp;
  private WindMeasurement wind;
  public Location loc;

  invariant temp == null || loc == temp.loc;
  invariant wind == null || loc == wind.loc;

  public void AddTemp(TempMeasurement! tm)
    requires loc == tm.loc;
    ensures loc == temp.loc;
    modifies this.temp;
  { temp = tm; }

  public void AddWind(WindMeasurement! wm)
    requires loc == wm.loc;
    ensures loc == wind.loc;
    modifies this.wind;
  { wind = wm; }
}

[Immutable]
public class Measurement {
  public Date date;
  public Location loc;
}

[Immutable]
public class TempMeasurement : Measurement {
  public Temp airTemp
  public Temp waterTemp;
}

[Immutable]
public class WindMeasurement : Measurement {
  public Velocity windVelocity;
}
```

Example 1

The main benefit of immutable classes is that the fields of object instances of such a class may be mentioned in invariants of any other object. Without immutable classes, to be allowed to mention the fields of another object in an invariant, this object has to be declared as [Rep]. But an object can only be declared as [Rep] by at most one other object. Therefore, this is no suitable solution, when an object should be mentioned in multiple objects' invariants.

The reason, why it cannot be allowed to mention arbitrary objects in an invariant, is that otherwise after each modification of an object, all classes would have to be checked for a potential invariant violation. This contradicts the principle of modular verification, which says that in order to verify the correctness of a class, one should not need to have access to all the code of classes that use or extend this class. A program verifier, therefore, cannot check every invariant of all classes after each modification of an object, as not all of those invariants might be known. However, this problem does not exist for immutable classes, because an object of such a class cannot be modified anyway, and no invariant can therefore be violated by an undetected modification of this object.

Instances of immutable classes are not allowed to be owned or have [Peer] fields. This implies that all those objects are peer consistent after they have been constructed, as they are always valid, have no owner, and their peers are guaranteed to be always consistent, because they have none. Hence, objects of an immutable type may always be used as receiver, parameter, or return value of a method call.

Example 1 shows a potential application for immutable classes. We have a class *WeatherReport* in which different kinds of measurements are stored. If we assume that once a measurement has been taken, the values of this measurement do not change anymore, then we can declare the class *Measurement* and its two subclasses *TempMeasurement* and *WindMeasurement* as immutable.

One restriction of immutable classes is that a class can only be declared as immutable, if its base class is *System.Object* or another immutable class. Conversely, that means that an immutable class cannot have mutable subclasses. Therefore, it would not be allowed to have a subclass of *Measurement* that is not immutable or to only declare *TempMeasurement* and *WindMeasurement* as immutable but not their base class *Measurement*.

The class *WeatherReport* contains a *Location*, a *TempMeasurement*, and a *WindMeasurement* field. It also has two invariants saying that as long as *temp* and *wind* are not *null*, their locations have to be the same as the one stored in field *loc* of the *WeatherReport* object. These invariants are only legal, because *Measurement* and its subclasses are declared as immutable. Otherwise, the *loc* fields of *temp* and *wind* would not be *visible* in those invariants, i.e. they could not be mentioned there, unless we would declare *temp* and *wind* as [Rep]. But as mentioned above, this would exclude the possibility of those *Measurement* objects being used by another class that also has an invariant depending on their fields.

2.3.2 Limitations

The concept of immutable classes is far from being perfect and has some severe limitations. For example, we might also want to have a class *CurrentWeather*, in which measurements are continually updated to always reflect the most current weather data:

```
public class CurrentWeather {
  private TempMeasurement temp;
  private WindMeasurement wind;

  public void UpdateTemp(Date d, Temp airT, Temp waterT)
  { ... }

  public void UpdateWind(Date d, Velocity windVel)
  { ... }
}
```

In this case, we need mutable instances of *TempMeasurement* and *WindMeasurement* and, therefore, cannot use the same *Measurement* class hierarchy that we defined in Example 1, because

all those instances are immutable. A first potential solution would be to use an immutable wrapper class that has an instance of the corresponding mutable class as a field:

```
[Immutable]
public class ImmutableMeasurement {
  private Measurement measurement;
}
```

```
public class Measurement {
  // implementation
}
```

However, we see that the two classes need to have a different name, despite being exactly the same aside from the [Immutable] modifier. From the compiler's point of view, the mutable and immutable instances are therefore not even of the same type, and we cannot, for example, assign an object of type *Measurement* to a field of type *ImmutableMeasurement* with this solution.

Another problem is the use of library classes. If we, for instance, want to have an immutable linked list, then not even a wrapper class can help, because the list would be immutable right after construction, and no elements could be added to it anymore, making it virtually useless. What we would need is the possibility to declare an object like this linked list as immutable only after a certain time and not already after its initialization, but this is not possible with the concept of immutable classes as implemented in Spec#.

A method *CorrectMeasurement* in class *Measurement*, that checks the validity of the different fields and fills in missing information if necessary, is another example where we would need an object to become immutable only after some time. Because if we want to call *CorrectMeasurement* every time before adding a measurement to the weather report, this is only possible if the object is still mutable at that point, which is not the case with immutable classes:

```
public void AddTemp(TempMeasurement! tm)
  requires loc == tm.loc;
  ensures loc == temp.loc;
  modifies this.temp;
{
  // not allowed: tm.CorrectMeasurement();
  temp = tm;
}
```

The concept of immutable classes offers no good solution for this problem, and so we either have to omit calling *CorrectMeasurement* and accept having potentially faulty data, or we have to leave the measurements mutable and, therefore, cannot write any invariants for their fields, or we could go back to using wrapper classes with the above mentioned disadvantages.

2.3.3 Implementation

The implementation of immutable classes is based on static type checks. In the *Prelude* file, those three functions declare whether a type is mutable or immutable:

```
function $IsImmutable(T:TName) returns (bool);
function $AsImmutable(T:TName) returns (theType: TName);
function $AsMutable(T:TName) returns (theType: TName);
```

Function *$IsImmutable* returns *true*, if the provided type is immutable. Function *$AsImmutable* returns the provided type again, if it is an immutable type, and *null* otherwise, while function *$AsMutable* is just the opposite.

With axioms, it is determined for each type, which functions hold for it. For classes *WeatherReport* and *Measurement* from Example 1, the following axioms would be added by Boogie while translating the program into BoogiePL:

```
axiom ¬$IsImmutable(WeatherReport) ∧ $AsMutable(WeatherReport) == WeatherReport;
axiom $IsImmutable(Measurement) ∧ $AsImmutable(Measurement) == Measurement;
```

With those axioms, a prover is able to check whether an object's type is immutable and has to be treated differently. Some operations have to be executed only when the type is mutable or, conversely, when it is immutable. On the one hand, assigning an owner to the object requires it to be mutable, as objects of an immutable type cannot be owned. On the other hand, it has to be checked for all invariants that when an object is not declared as [Rep], its fields are only mentioned in the invariant, when the type of the object is immutable. This introduces many additional checks that have to be executed by a prover whether the program has immutable classes or not.

2.4 Frozen Objects

The frozen objects methodology is a concept designed to overcome the restrictions of immutable classes and has been introduced by Leino, Müller, and Wallenburg [7]. They have shown, how this technique can work based on a simplified Spec#-like language. Their paper also describes the theoretical background of the methodology in detail.

In this section, we introduce the methodology and compare it with immutable classes. In the following chapters, we then describe, how we implemented this technique into Spec# and extended the verifier Boogie with the capabilities to verify programs using frozen objects.

2.4.1 General Concept

The main idea of frozen objects is to provide immutability on an object level instead of on a class level as with immutable classes. Declaring an object as immutable (called *frozen*) should be as easy as assigning an owner to an object. The frozen objects methodology, thus, introduces a new object called *Freezer*, with the intention that every object directly or transitively owned by this *Freezer* is frozen.

To achieve this, the *Freezer* object is not allowed to be referred to by the rest of the program. As there are no references to it, the *Freezer* in particular can never be exposed by a programmer and, hence, is always valid.

If the *Freezer* object is assigned as the owner of another object, this object cannot be exposed anymore either from that point on, because exposing an object requires its owner (if it has any) not to be valid, which would contradict the ever validity of the *Freezer*. And this applies not only to objects directly owned by the *Freezer*, but also to objects transitively owned by it. The reasoning is exactly the same, namely that their respective owner is always valid, because either it is the *Freezer* object, or an object that is itself (transitively) owned by the *Freezer*.

With this technique, one can thus declare any object as frozen by just assigning the *Freezer* as its owner (called *freezing* the object). The keyword *freeze* executes exactly this operation, i.e. the *Freezer* is assigned as new owner of the provided object.

The keyword *frozen* can be put before a field or method parameter declaration. A field or parameter specified as *frozen* may only hold *null* or a reference to a frozen object, but not a reference to a mutable object. The following small example shows how to use those two keywords:

```
public class Foo {
  frozen Foo f = null;

  public void Bar() {
    Foo temp = new Foo();
    freeze temp;
    f = temp;
  }
}
```

We first create a local variable *temp* of type *Foo*, then call *freeze* on this variable, and finally assign it to the field *f* that has been declared with the *frozen* keyword.

2.4.2 Benefits over Immutable Classes

The frozen objects methodology solves many limitations of immutable classes. There is no more restriction in subclassing, there can be mutable and immutable object instances of the same type, objects from library classes can also easily be frozen, and objects can be frozen at any point in time, not only just after they have been initialized.

Because fields declared as *frozen* are guaranteed to never be modified, an invariant depending on their state cannot be violated anymore, once it has been established. Writing such invariants is therefore allowed, providing frozen objects with the same advantage as objects of an immutable type.

Example 2 shows the same classes as Example 1, just without declaring them as immutable. Apart from that, *Measurement* and its subclasses stay exactly the same. Instead of the [Immutable] modifier before the classes, we add the *frozen* keyword before the declaration of each field that we want to be frozen. This allows us to keep the invariants mentioning the fields *temp* and *wind*.

```
public class WeatherReport {
  frozen private TempMeasurement temp;
  frozen private WindMeasurement wind;
  public Location loc;

  invariant temp == null || loc == temp.loc;
  invariant wind == null || loc == wind.loc;

  public void AddTemp(frozen TempMeasurement! tm)
    requires loc == tm.loc;
    ensures loc == temp.loc;
    modifies this.temp;
  { temp = tm; }

  public void AddWind([Captured] WindMeasurement! wm)
    requires loc == wm.loc;
    ensures loc == wind.loc;
    modifies this.wind;
  {
    freeze wm;
    wind = wm;
  }
}

public class Measurement {
  public Date date;
  public Location loc;
}

public class TempMeasurement : Measurement {
  public Temp airTemp
  public Temp waterTemp;
}

public class WindMeasurement : Measurement {
  public Velocity windVelocity;
}
```

Example 2

The *frozen* keyword before parameter *tm* in method *AddTemp* requires that the provided *TempMeasurement* object has to be already frozen. Therefore, we can just assign this object to the frozen field *temp*.

Another possibility is shown by the method *AddWind*, where we do not get an already frozen object. Hence, we freeze the provided object first, before assigning it to the frozen field *wind*. Because the owner of *wm* is changed by the *freeze* keyword, the parameter has to be declared with the [Captured] modifier, as only the owner of a captured parameter is allowed to be modified inside a method.

A first benefit we get in this example for using frozen objects, is that we are now allowed to modify an object between its construction and the time when the object is frozen. For example, we can now have a method *CorrectMeasurement*, which checks the validity of all the fields and fills in missing information for fields that have not been initialized with a meaningful value:

```
public virtual void CorrectMeasurement()
  ensures loc == old(loc);
{
  // check validity of measurement and fill in missing information
}
```

We define this virtual method in the class *Measurement* and then override it in each subclass to additionally check the newly defined fields in that class. With immutable classes, such a method has been useless, because the objects were already immutable after their initialization, and values could therefore not be corrected anymore. With frozen objects, however, we may call *CorrectMeasurement* before freezing the *wm* object in method *AddWind*:

```
public void AddWind([Captured] WindMeasurement! wm)
  requires loc == wm.loc;
  ensures loc == wind.loc;
  modifies this.wind, wm.*;
{
  wm.CorrectMeasurement();
  freeze wm;
  wind = wm;
}
```

At the time *CorrectMeasurement* is called, *wm* is still mutable and the method call therefore perfectly legal. The postcondition in *CorrectMeasurement* ensures that a call to this method does not invalidate the invariant 'wind == null || loc == wind.loc' of class *WeatherReport*. Without this postcondition, Boogie would detect a possible invariant violation when assigning *wm* to *wind*.

We cannot do the same in method *AddTemp*, where we already get a frozen object as parameter. In that case, we have to call *CorrectMeasurement* already earlier, before freezing the *tm* object and calling *AddTemp*:

```
WeatherReport wr = new WeatherReport(loc);
TempMeasurement tm = new TempMeasurement(date, loc, airT, waterT);
tm.CorrectMeasurement();
freeze tm;
wr.AddTemp(tm);
```

Another benefit of frozen objects is that we may also have a class *CurrentWeather* that continually updates the measurements without ever wanting them to become frozen, but still uses the same measurement type hierarchy, which has not been possible with immutable classes:

```
public class CurrentWeather {
  [Rep] private TempMeasurement! temp;
  [Rep] private WindMeasurement! wind;
```

```
  public CurrentWeather(Date d, Location l) {
    temp = new TempMeasurement(d, l, null, null);
    wind = new WindMeasurement(d, l, null);
  }

  public void UpdateTemp(Date d, Temp airT, Temp waterT) {
    expose (this) {
      temp.date = d;
      temp.airTemp = airT;
      temp.waterTemp = waterT;
    }
  }

  public void UpdateWind(Date d, Velocity windVel) {
    expose (this) {
      wind.date = d;
      wind.windVelocity = windVel;
    }
  }
}
```

We declare the fields *temp* and *wind* as [Rep]. In methods *UpdateTemp* and *UpdateWind*, we may update those fields by exposing the *this* object reference, because *this* is the owner of both fields.

All the benefits listed in this section make frozen objects much more flexible to use than immutable classes in most application scenarios. In the following chapters, we explain how we transferred this methodology from the simplified language of the original paper [7] to Spec#, and how we extended Boogie, so that we may verify properties of frozen objects with this static program verifier.

Chapter 3

Extending Spec#

Adding the frozen objects methodology to Spec# is split in three main parts. First, we have to extend the Spec# compiler to add the new keyword and modifiers required by the methodology. Next, we have to encode the theory of frozen objects in the *Prelude* file. And finally, we also have to extend the Boogie verifier in order to be able to verify programs using frozen objects.

In this chapter, we describe in detail the modifications in the source code of Spec# that have been necessary to implement the frozen objects methodology. The modifications for *Prelude* and Boogie are explained in the next two chapters.

3.1 Characteristics of Spec#

We begin by pointing out the differences between Spec# and the simplified language used in the paper introducing frozen objects [7]. Spec# is a more complex language and has several characteristics that make the implementation of the frozen objects methodology more challenging. We discuss the implications of those characteristics in this section, before continuing with the actual implementation.

In the simplified language, assigning the *Freezer* as the owner of any object is indeed a straight forward way of declaring that object as frozen. This only works in every case, however, because ownership transfer is allowed in that language.

Spec# does not support ownership transfer. Therefore, objects that already have an owner cannot be frozen anymore. On the one hand, this means that no [Peer] or [Rep] object may be frozen. On the other hand, it guarantees that once an object is frozen, it can never be unfrozen again, because that, too, would require an ownership transfer.

We have to be careful what being frozen exactly means in Spec#. If we freeze an object, then only those fields that are declared as either [Peer] or [Rep] also get frozen. [Peer] fields have the same owner as the object in which they are declared. So if we assign the *Freezer* as owner of an object, then also all of its peers have to be frozen. [Rep] objects are owned by the object in which they are declared, and after freezing this object, all [Rep] fields are thus transitively owned by the *Freezer* as well.

However, we may also have fields that are not specified as [Peer] or [Rep] in Spec#. Such fields have no ownership relation to the object they are contained in. Hence, even if the object containing a non-peer, non-rep field gets frozen, this field is not guaranteed to be transitively owned by the *Freezer* afterwards and may still be mutable.

A non-peer, non-rep field cannot be modified via an assignment 'frozenObj.field = newValue', because this would require the target object to be peer consistent. However, a frozen object can never be peer consistent, as that would require its owner to be exposed. Therefore, it is guaranteed that the object reference of a non-peer, non-rep field always stays the same in a frozen object. Nevertheless, there can be aliases pointing to the same object, and over such an alias, we may still modify the state of an object referenced by a non-peer, non-rep field. So we have to be aware that

freezing an object in Spec# not necessarily ensures the immutability of every object referenced by it. This can be seen here:

```
public class Foo {
  public A! af;

  public Foo(A! a) {
    af = a;
  }

  public void Bar() {
    A a = new A();
    Foo foo = new Foo(a);
    freeze foo;

    // not allowed: foo.af = new A();

    // allowed, because 'a' is not transitively owned by the Freezer
    a.i = 4;
  }
}

public class A {
  public int i;
}
```

In method *Bar*, we create an object *a* of type *A*, give it to the constructor of *Foo*, which initializes its field *af* with it, and then freeze object *foo*. Afterwards, we are not allowed anymore to assign a new *A* object to field *af* of *foo*. However, we may still modify the integer field *i* of object *a*, even though it is the same object as referenced by the field *af* of frozen object *foo*. This is only possible, because *a* is still unowned and not owned by the *Freezer* like *foo*.

If field *af* would have been declared as [Peer] or [Rep], it would not be possible to modify *a* at this point in the code anymore, because it would be directly or transitively owned by the *Freezer*, making it impossible to satisfy the requirement of this assignment statement that the owner of *a* has to be exposed first.

3.2 Freeze Keyword and IsFrozen Check

Starting with the implementation of the methodology, we first have to add the new *freeze* keyword to the Spec# compiler which, amongst some other new definitions, includes adding the following statement to the *Scanner* class:

```
keyword = new Keyword(Token.Freeze, "freeze", true, keyword);
```

This allows the compiler to recognize the string "freeze" in a program as a keyword during the scanning process. In the *Parser* class, we then add a new method *ParseFreeze* that gets called, whenever the compiler encounters a use of *freeze*. This method creates a new *Freeze* object and sets the *Expression* field of that object to the value returned by parsing everything between the *freeze* keyword and the next semicolon.

After being scanned and parsed by the Spec# compiler, an input file next goes through a number of different visitor classes. For the new *freeze* keyword, we therefore have to add a new *VisitFreeze* method to all these classes. Most of those implementations are straight forward, and we mention here only the two, where actually something interesting happens.

The first is in class *Checker*, where the compiler verifies that the *Freeze* object has an expression associated with it, and that this expression is a valid object reference:

```
public override Statement VisitFreeze(Freeze Freeze) {
  if (Freeze == null) return null;
  if (Freeze.Expression == null || (Freeze.Expression.Type != null &&
      Freeze.Expression.Type.IsAbstract))
    this.HandleError(Freeze, Error.ObjectRequiredForFreeze);

  Freeze.Expression = this.VisitExpression(Freeze.Expression);
  return Freeze;
}
```

With these checks, the following two illegal uses of *freeze* get detected, and an error is thrown in both cases. The *freeze* keyword in *Foo* has no expression following it, and the expression following *freeze* in *Bar* is a type name and not an object reference as is expected:

```
public class Test {
  public void Foo() {
    freeze;
  }

  public void Bar() {
    freeze Test;
  }
}
```

The second interesting *VisitFreeze* method is the one in class *Normalizer*, where the *freeze* statement is turned into a method call, which is responsible for the actual freezing of the associated object reference:

```
public override Statement VisitFreeze(Freeze Freeze) {
  Method freezeMethod = this.GetTypeView(SystemTypes.FreezeHelpers).GetMethod(
      Identifier.For("Freeze"), SystemTypes.Object);
  Freeze.Expression = this.VisitExpression(Freeze.Expression);
  MethodCall mc = new MethodCall(new MemberBinding(null, freezeMethod),
      new ExpressionList(Freeze.Expression), NodeType.Call,
      SystemTypes.Void, Freeze.SourceContext);
  Freeze.Expression = mc;
  return Freeze;
}
```

First, method *Freeze* of class *FreezeHelpers* is assigned to a new *Method* object *freezeMethod*. Then a new *MethodCall* object *mc* is created, which calls the *freezeMethod* with the value of the *Freeze.Expression* field as a parameter and *void* as return value. This *MethodCall* object is then assigned as new value of the *Freeze.Expression* field, meaning that the field now does not contain the object reference to the object we want to freeze anymore, but instead a call to the method responsible for the freezing with the object reference as a parameter to it.

The above mentioned new *FreezeHelpers* class contains only two static methods. The first is responsible for freezing the provided object, while the second returns a boolean value, indicating whether the given object is frozen:

```
public class FreezeHelpers {
  /// As seen by the static program verifier: Requires that subject initially
  /// has no owner, and sets the owner of subject to ($freezerRef, $Freezer).
  /// Dynamically, this method is a no-op.
  public static void Freeze([NotNull] [Delayed] object subject) { }
```

```
  /// As seen by the static program verifier: returns true iff subject
  /// is owned by ($freezerRef, $Freezer).
  /// Dynamically, this method always returns true.
  [Pure] [Reads(ReadsAttribute.Reads.Everything)]
  public static bool IsFrozen([Delayed] object subject) { return true; }
}
```

However, both of those methods have no implementation (the second simply returns *true*). Hence, on normal execution of a program, they do nothing at all, because ownership and, therefore, also freezing of objects is only relevant, when trying to prove properties of a program with the static program verifier Boogie, but not when simply executing the program. We explain in Chapter 5, how Boogie takes those method calls and replaces them by an actual implementation while translating a program into BoogiePL.

The *IsFrozen* method may be used in assertions to verify whether a given object is frozen at any point in the program:

```
assert FreezeHelpers.IsFrozen(obj);
```

It may also be used in invariants, pre- and postconditions, but the [Frozen] modifier we introduce in the next section is much better suited for that purpose.

3.3 [Frozen] Modifier

To define which fields of an object should be frozen, we introduce a [Frozen] modifier with which a field can be specified as follows:

```
[Frozen] TempMeasurement temp;
```

This [Frozen] modifier replaces the *frozen* keyword from the original paper [7], because [Frozen] is very similar to the [Peer] and [Rep] modifiers, as all three say something about the owner of the field they are attached to. In order to keep a certain consistency, we therefore decided to make [Frozen] a modifier as well and not a keyword.

A field that is declared with this [Frozen] modifier can be either *null* or contain a reference to a frozen object. But it is not allowed to contain a reference to an object that is not frozen. In order to guarantee that this condition always holds, whenever an assignment to a field declared [Frozen] takes place, it is checked whether the object that gets assigned to the field is already frozen. If it is not, then the object gets frozen during the assignment. This constitutes a case of implicitly freezing an object, compared to the explicit freezing with the *freeze* keyword, where the programmer has to type in an extra statement in order to freeze an object.

This implicit freezing allows us to simplify the *AddWind* method of the *WeatherReport* class that we have seen before:

```
public class WeatherReport {
  [Frozen] private WindMeasurement wind;

  public void AddWind([Captured] WindMeasurement! wm)
    modifies this.wind;
  {
    // not needed: freeze wm;
    wind = wm;
  }
}
```

It is not necessary in this case to explicitly call 'freeze wm' before assigning *wm* to the frozen field *wind*, because the freezing of the object takes place implicitly during the assignment. If we

explicitly freeze the object first, then it is detected during the assignment that the object is already frozen and, thus, nothing has to be done anymore.

The [Frozen] modifier may not only be used on fields, but also to declare a parameter or return value of a method as frozen. The method *AddTemp* of class *WeatherReport* uses the [Frozen] modifier before parameter *tm* to ensure that only frozen objects are passed to the method. We could also add a method *GetTemp* to the class that returns the *temp* field. To guarantee that this method always returns a frozen object, we define the return value as frozen:

```
public class WeatherReport {
  [Frozen] private TempMeasurement temp;

  public void AddTemp([Frozen] TempMeasurement! tm)
    modifies this.temp;
  { temp = tm; }

  [return: Frozen]
  public TempMeasurement GetTemp() {
    return temp;
  }
}
```

The [Frozen] modifier has been defined as a subclass of *Attribute* same as modifiers [Peer] and [Rep]. The [AttributeUsage] modifier is used to define that [Frozen] is only allowed before a field, a method parameter, or a return value:

```
[AttributeUsage(AttributeTargets.Field | AttributeTargets.Parameter |
    AttributeTargets.ReturnValue, AllowMultiple = false, Inherited = false)]
public sealed class FrozenAttribute : Attribute { }
```

We added some additional checks to the Spec# compiler. It is not allowed to declare a field with the [Peer] or [Rep] modifier in addition to [Frozen], as that would require the field to have two different owners at the same time. A field, parameter, or return value of a value type may also not be declared as [Frozen].

A method parameter with the [Inside] modifier, which requires this parameter to already have been exposed by the caller of the method, may not additionally be specified as [Frozen], because an object cannot be frozen and exposed at the same time. Neither can a parameter be declared as [Captured] and [Frozen] at the same time, because the first would require the object to be unowned and the second that the object is transitively owned by the *Freezer*.

For virtual methods with a return value declared [Frozen], we have to make sure that all methods overriding it also have this same modifier. We expect to get a frozen object back, if we call this virtual method. But if the object at runtime is actually of a subtype, and a method overriding this virtual method without specifying the return value as [Frozen] is called, the object we get back would not necessarily be frozen.

With the new frozen objects methodology, we allow fields of frozen objects to be mentioned in invariants. Hence, we have to modify the implementation in a way that no compiler error is thrown on access of a frozen object's field in an invariant of any other object.

If we declare an array with the [Frozen] modifier, then it is only ensured that the references to the array elements stay the same, i.e. we cannot add or remove any elements from such an array. But the array elements themselves are still mutable. This is similar to non-peer, non-rep fields of a frozen object. To which object such a field refers to cannot be modified, but the object's state can still be altered, because it is not owned transitively by the *Freezer*. Arrays also do not own their elements and that is why the elements of an array are still mutable, even if the array is declared as frozen.

3.4 [ElementsFrozen] Modifier

We introduce an [ElementsFrozen] modifier (similar to the already existing [ElementsPeer] and [ElementsRep] modifiers) for declaring the elements of an array as frozen. This modifier has no equivalent in the paper introducing the frozen objects methodology [7], as arrays have not been part of the simplified language used there.

Array fields may be specified with this [ElementsFrozen] modifier, guaranteeing that all objects contained in the array are either *null* or frozen. This is independent from the fact whether the array itself is frozen. Therefore, we can define three different types of immutability on arrays:

```
[Frozen] object[] a1;
[ElementsFrozen] object[] a2;
[Frozen] [ElementsFrozen] object[] a3;
```

The array *a1* is frozen and, hence, no elements can be added or removed from it, but the elements themselves are still mutable. Array *a2* is the exact opposite, because its elements are declared as frozen, but we can still add elements to or remove them from the array, as long as the added elements are frozen (or can implicitly be frozen during the assignment). Array *a3* combines both by being frozen and also having only frozen elements.

In addition, the [ElementsFrozen] modifier may also be used in combination with generic types, where we can declare one or all type parameters as frozen. If we, for example, have a class *Dictionary* with two generic type parameters, there are the following options for using the [ElementsFrozen] modifier:

```
[ElementsFrozen(0)] Dictionary<Foo, Bar> d1;
[ElementsFrozen(1)] Dictionary<Foo, Bar> d2;
[ElementsFrozen] Dictionary<Foo, Bar> d3;
```

In the first example, the [ElementsFrozen(0)] modifier declares the first type parameter of *d1* as frozen, meaning that wherever this parameter is used in class *Dictionary*, not only an object of type *Foo* is expected, but more specifically a frozen object of this type. For object *d2* the second type parameter is declared as frozen, while no specific type parameter is targeted by the [ElementsFrozen] modifier of *d3*, meaning that both are specified as frozen.

We may additionally add the [Frozen] modifier to each of the *Dictionary* objects, as seen with the array example above, thus even further increasing the various options of declaring parts of objects as frozen, while leaving other parts potentially mutable.

The [ElementsFrozen] modifier has been implemented as a subclass of *AttributeWithContext*. Fields, parameters, and return values can be declared with this modifier, same as for [Frozen]:

```
[AttributeUsage(AttributeTargets.Field | AttributeTargets.Parameter |
    AttributeTargets.ReturnValue, AllowMultiple = true, Inherited = false)]
public sealed class ElementsFrozenAttribute : AttributeWithContext {
  public int Value;
  public ElementsFrozenAttribute() { this.Value = -1; }
  public ElementsFrozenAttribute(int value) { this.Value = value; }
}
```

In contrast to [Frozen], the [ElementsFrozen] modifier can be used multiple times before a declaration. For example, if we have a field with three generic type parameters, and we want to declare the first two of them as frozen, we need to add two [ElementsFrozen] modifiers before the field declaration:

```
[ElementsFrozen(0)] [ElementsFrozen(1)] Foo<S, T, U> foo;
```

As with [Frozen], if we assign an object to a field with some or all elements declared as [ElementsFrozen], then the freezing of those elements takes place implicitly, if they are not already

frozen. The problem of how to allow a method to modify the owner of an object's elements is addressed in the next section.

We added some additional checks to detect wrong usage of the new modifier already on compile time. [ElementsFrozen] may only be attached to fields, parameters, and return values, if the type is an array or a subtype of *IEnumerable* or an *IEnumerable*-like type. We also have to check that a method overriding another method with a return value specified as [ElementsFrozen] keeps this definition for the same reason as with [Frozen].

3.5 [ElementsCaptured] Modifier

The [Captured] modifier in Spec# requires a parameter specified with it to be unowned. Inside the method, it is then allowed to assign an owner to such a parameter. We have seen earlier that if we want to freeze an object provided as method parameter or assign this parameter to a frozen field, we have to declare the parameter as [Captured], because freezing is just another way of assigning an owner to an object.

However, the [Captured] modifier does not allow us to modify the owner of an object's elements. This leads to the problem of not being able to assign a method parameter to a field declared with the [ElementsFrozen] modifier, unless the corresponding elements of the object are already frozen when calling the method. But if we want to modify an object's elements in a method before assigning the object to the field specified as [ElementsFrozen], we need a possibility of allowing a method to modify the owners of an object's elements.

For this purpose, we introduce the [ElementsCaptured] modifier. This modifier requires the targeted elements of an object to be unowned and allows the owner of those elements to be modified inside a method. As with the [ElementsFrozen] modifier, [ElementsCaptured] may target all the elements of an array or a generic type, or it may only declare one or more specific type parameters as capturable.

```
public class Foo {
  [Peer] [ElementsFrozen] Foo[] fa;
  [ElementsFrozen(0)] Dictionary<object, string> dic;

  public void Bar1([Captured] [ElementsCaptured] Foo[]! fa) {
    this.fa = fa;
  }

  public void Bar2([ElementsCaptured(0)] Dictionary<object, string>! dic) {
    this.dic = dic;
  }
}
```

In the above example, we show how to use the [ElementsCaptured] modifier. Method *Bar1* assigns an array object of type *Foo* to the field *fa* that is declared as [Peer] and [ElementsFrozen]. When assigning the method parameter to *fa*, an owner is not only assigned to the array but also to its elements. Therefore, we need to specify the method parameter as [Captured] as well as [ElementsCaptured].

Method *Bar2* takes an object of the generic type *Dictionary* as a parameter and assigns it to the field *dic* whose first type parameter is declared as frozen. We therefore need to attach a corresponding [ElementsCaptured(0)] modifier to the method parameter in order to allow the assignment to field *dic*. Usually, just using the [ElementsCaptured] modifier without an attached value would also be an option, but not in this specific case where the second type parameter is of type *string*. Strings are of an immutable type, implying that with the new frozen objects methodology, all strings are frozen (see Chapter 6.4). Hence, strings can never be unowned as would be required by using the [ElementsCaptured] modifier without an attached value and, thus, targeting both type parameters.

The definition of this modifier is similar to the one for [ElementsFrozen], in that it can be defined with or without an integer value. [ElementsCaptured] may not only be applied to method parameters but also to methods and constructors, in which case the target of the modifier is the receiver object.

As with [ElementsFrozen], the compiler checks that only arrays or subtypes of *IEnumerable* or an *IEnumerable*-like type can be specified as [ElementsCaptured]. We additionally have to ensure that every method overriding a virtual method declared as [ElementsCaptured] also has this modifier attached to it.

Chapter 4

Axiomatization of Frozen Objects

In this chapter, we explain our formalization of the frozen objects methodology. We add this axiomatization to the *Prelude* file that contains the background theory of Spec#. In Chapter 5, we then explain how Boogie uses this axiomatization to verify programs that include frozen objects.

4.1 Freezer

We introduce a *Freezer* object that exists only in the context of the BoogiePL language without a counterpart in the Spec# code. We represent this *Freezer* with two new constants as opposed to only one in the formal encoding of frozen objects in the original paper [7]. The reason for using two constants, is that an owner in Spec# is defined by an (object, class frame) tuple.

The first constant we introduce is of type *TName*, which stands for a type declaration, and defines the type *$Freezer*. The second constant is of type *ref* and defines an object reference:

```
const unique $Freezer : TName;

const unique $freezerRef : ref;
```

That the *$freezerRef* object reference is of type *$Freezer* can be encoded with the following simple axiom:

```
axiom $typeof($freezerRef) == $Freezer;
```

However, this axiom is not really needed in order for the axiomatization to be sound. It may therefore also be omitted, because it just serves the purpose of explaining the connection between the two constants.

4.2 Freezing an Object

Whenever an object - whether explicitly through the *freeze* keyword or implicitly during an assignment - has to be frozen, the procedure *$UpdateOwnersForFrozen* is called to take care of assigning the *Freezer* as the owner of the provided object:

```
procedure $UpdateOwnersForFrozen(x: ref);
  modifies $Heap;
  ensures (∀<alpha> p: ref, F: Field alpha • (F ≠ $ownerRef ∧ F ≠ $ownerFrame ∧
    F ≠ $FirstConsistentOwner) ∨ old($Heap[p, $ownerRef] ≠ $Heap[x, $ownerRef] ∨
    $Heap[p, $ownerFrame] ≠ $Heap[x, $ownerFrame]) ⇒
    old($Heap[p, F]) == $Heap[p, F]);
  ensures x == null ⇒ $Heap == old($Heap);
```

```
ensures x ≠ null ⇒ (∀ p: ref • old($Heap[p, $ownerRef] == $Heap[x, $ownerRef]
  ∧ $Heap[p, $ownerFrame] == $Heap[x, $ownerFrame]) ⇒
  $Heap[p, $ownerRef] == $freezerRef ∧ $Heap[p, $ownerFrame] == $Freezer);
free ensures $HeapSucc(old($Heap), $Heap);
```

This procedure is similar to procedures *$UpdateOwnersForPeer* and *$UpdateOwnersForRep* which are already in *Prelude* and assign the correct owner to an object specified as [Peer] or [Rep].

$UpdateOwnersForFrozen takes an object reference as input and, if it is *null*, ensures through the second postcondition that all objects remain unaltered. If the provided object is not *null*, the third postcondition ensures that the *$ownerRef* of the object and all its peers is set to *$freezerRef* and the *$ownerFrame* of all those objects is set to the type *$Freezer*.

The reason why we also need to freeze all peers, is that they always need to have the same owner. But if such a peer group already has an owner, then we cannot freeze any of its objects anymore. The only way how a group of objects may be frozen together, is when they are in a special peer group with an owner tuple (x, $PeerGroupPlaceholder), where x is a representative object of the peer group. If we have such a peer group of unowned objects, then we guarantee that if one of those objects gets frozen with the *$UpdateOwnersForFrozen* procedure, all the others get frozen as well.

What the first postcondition expresses, is that all other objects apart from the *$ownerRef* and *$ownerFrame* fields of the target object and its peers remain unchanged. Without this postcondition, a prover could not be sure whether any other object has been modified by the procedure and, therefore, could not assert certain facts anymore after a call to *$UpdateOwnersForFrozen*.

The last postcondition is a free one, meaning that a prover can just assume it to hold and does not have to assert it. It just states that the heap at the end of the procedure is a successor of the heap at the beginning of it. Each procedure has such a free postcondition and we explain in Chapter 4.6 for what it is used.

4.3 Frozen Objects

To determine whether an object is frozen, we add a function *IsFrozen* that takes an object reference and a heap as inputs and returns a boolean value, indicating if the given object is frozen in this heap:

```
function IsFrozen(o: ref, h: HeapType) returns (bool);
```

We also add a new boolean ghost field called *$frozen* to the axiomatization and state in a simple axiom that whenever the *IsFrozen* function holds for an object, the *$frozen* field of this object is set to *true* and vice versa:

```
const unique $frozen : Field bool;

axiom (∀ h: HeapType, o: ref • IsHeap(h) ∧ IsFrozen(o, h) ⇔ h[o, $frozen]);
```

This additional field often makes it easier for a prover to determine whether an object is frozen, because it is difficult for the prover to assert that the return value of the *IsFrozen* function has not changed after some assignment statements in a method. With this new *$frozen* field, however, it is clear to a prover that as long as this field is not modified, an object that has been frozen before an assignment statement is still frozen after this statement. As access to a field of an object is faster than evaluating a function, we use the *$frozen* field instead of the *IsFrozen* function in the antecedents of the following axioms.

Writing an axiom directly stating that every object transitively owned by the *Freezer* is frozen results in bad performance for program verification, because automatic theorem provers perform poorly on transitive closure [7]. The following two axioms encode this transitive closure in a slightly different way, resembling the concept of mathematical induction:

```
axiom (∀ h: HeapType • IsHeap(h) ⇒ IsFrozen($freezerRef, h));

axiom (∀ h: HeapType, o: ref • IsHeap(h) ∧ o ≠ null ∧
    h[h[o, $ownerRef], $frozen] ⇒ IsFrozen(o, h));
```

The first axiom says that for all heaps, the *$freezerRef* object reference is frozen. The second axiom states that if the owner of an object is frozen, then the object is frozen itself. Hence, by taking both those axioms into account, one can infer for every object transitively owned by the *Freezer* that it is frozen.

In particular, these two axioms allow a prover to assert the fact that an object referred to by a [Rep] field of a frozen object is also frozen. It is also clear that the same should hold for a [Peer] field, as the two peers share the same owner and are therefore either both frozen or mutable. But this cannot be inferred solely by the two axioms above, because it has to be assumed that there might be other ways, how an object could be declared as frozen. We could rewrite the second axiom to go both ways, i.e. also expressing the fact that if an object is frozen, so is its owner. But we realized that such a change results in poor performance for the verification of many programs. Instead, we add the following axiom:

```
axiom (∀ h: HeapType, o: ref, p: ref • IsHeap(h) ∧ h[o, $frozen] ∧
    p ≠ null ∧ h[p, $ownerRef] == h[o, $ownerRef] ∧
    h[p, $ownerFrame] == h[o, $ownerFrame] ⇒ IsFrozen(p, h));
```

This axiom simply states that if an object is frozen, so are all other objects with the same (object, class frame) tuple as owner, making it easy for a prover to infer that a [Peer] field of a frozen object is also frozen.

4.4 Frozen Fields

To encode that certain fields of an object are declared as [Frozen] or [ElementsFrozen], we introduce the following two functions:

```
function AsFrozenField(f: Field ref) returns (theField: Field ref);

function AsElementsFrozenField(f: Field ref, position: int) returns
    (theField: Field ref);
```

The first function takes a field reference as argument and returns the same field again, if it has been specified with the [Frozen] modifier, or *null* otherwise. The second function takes a field reference and an integer as arguments and returns the same field again, if it has been declared with the [ElementsFrozen] modifier for the given position, or *null* otherwise. The position attribute is -1 for arrays, which has the effect that all elements in the array are frozen. For generic types, the corresponding position of the type parameter (starting with 0) is used to declare all occurrences of this parameter as frozen.

Both of the above functions are analogous to the functions *AsPeerField*, *AsRepField*, *AsElementsPeerField*, and *AsElementsRepField* in order to maintain a certain consistency in the encoding, and because they serve an almost identical purpose.

The functions *AsFrozenField* and *AsElementsFrozenField* return *null*, whenever no axiom is saying otherwise. Hence, for each field declared as [Frozen] or [ElementsFrozen] corresponding axioms have to be added. For example, for those four objects:

```
[Frozen] object[] a1;
[ElementsFrozen] object[] a2;
[Frozen] [ElementsFrozen] object[] a3;
[ElementsFrozen] Dictionary<Foo, Bar> d;
```

the following axioms are added automatically to the BoogiePL code, when Boogie translates a Spec# source file with the above field declarations in it:

```
axiom AsFrozenField(a1) == a1;
axiom AsFrozenField(a3) == a3;
axiom AsElementsFrozenField(a2, -1) == a2;
axiom AsElementsFrozenField(a3, -1) == a3;
axiom AsElementsFrozenField(d, 0) == d;
axiom AsElementsFrozenField(d, 1) == d;
```

The first two axioms state that arrays *a1* and *a3* are frozen. The next two axioms encode that the elements of arrays *a2* and *a3* are frozen, and the last two axioms declare the two generic type parameters of *d* as frozen.

We now need two more axioms encoding that all objects for which the above functions do not return *null* are frozen. Hence, the first axiom states that a field declared as [Frozen] is indeed frozen, and the second that all the elements of an array or all occurrences of a type parameter specified as [ElementsFrozen] are frozen:

```
axiom (∀ h: HeapType, o: ref, f: Field ref • (IsHeap(h) ∧
    h[o, AsFrozenField(f)] ≠ null) ⇒ IsFrozen(h[o, AsFrozenField(f)], h));

axiom (∀ h: HeapType, o: ref, f: Field ref, i: int •
    IsHeap(h) ∧ h[o, AsElementsFrozenField(f, i)] ≠ null ⇒
    IsFrozen($ElementProxy(h[o, AsElementsFrozenField(f, i)], i), h));
```

In the second axiom, we use an object called *$ElementProxy* that represents all the elements of an array or all the occurrences of a type parameter in a generic class respectively. Thus, if for a field *f* at position *i* the *AsElementsFrozenField* function does not return *null*, the axiom just has to ensure that the *$ElementProxy* object of field *f* is frozen at position *i*.

4.5 Peer Validity

A frozen object is always peer valid, because it can never be exposed, and its peers are also frozen and, hence, unexposable. We express this in the following axiom:

```
axiom (∀ h: HeapType, o: ref • IsHeap(h) ∧ h[o, $frozen] ⇒
    (∀ p: ref • p ≠ null ∧ h[p, $allocated] ∧
    h[p, $ownerRef] == h[o, $ownerRef] ∧ h[p, $ownerFrame] == h[o, $ownerFrame]
    ⇒ h[p, $inv] == $typeof(p) ∧ h[p, $localinv] == $typeof(p)));
```

Without this axiom, it would not be possible to call a pure method on a frozen object or to pass a frozen object as parameter to a pure method. However, this should both be allowed, as pure methods ensure that they do not modify their receiver and parameters and, therefore, cannot violate the immutability of frozen objects.

4.6 Forever Frozen

Leino, Müller, and Wallenburg have suggested the following axiom in their paper [7], stating that if an object is frozen in a certain heap, then this object is still frozen in each successor heap, in order to encode that an object that has once been frozen stays frozen forever:

```
axiom (∀ oldHeap: HeapType, newHeap: HeapType, o: ref •
    $HeapSucc(oldHeap, newHeap) ∧ oldHeap[o, $frozen] ⇒ IsFrozen(o, newHeap));
```

The above axiom only works properly, if it is clearly defined that each heap occurring in a Boogie program is a successor of all the heaps before it. Thus, every procedure has to ensure that the heap at the end of the procedure is a successor of the heap at the beginning of it. This explains why we had to add the free postcondition '$HeapSucc(old($Heap), $Heap)' to the procedure *$UpdateOwnersForFrozen*.

This axiom is, however, not necessarily needed in Spec#, because in contrast to the simplified language from the original paper, Spec# does not allow ownership transfer. Hence, an object transitively owned by the *Freezer* is forever transitively owned by it. And even if ownership transfer were allowed, being frozen still ensures that no fields of an object can be modified, including the owner field of the object.

Despite this already given double security that a frozen object always stays frozen, we decided to still add the above axiom, because a prover only acts in very simple steps. If some assertion cannot be verified by the prover, then it just assumes that this assertion does not hold. With this axiom, we provide a simple way of keeping track of frozen objects by saying that if an object has been proven as frozen once, this assumption holds in all following heaps of the program execution.

Another reason is the performance, because even though a prover could in most cases come to the same conclusions without the help of the above axiom, it would often just take much longer to do so.

4.7 Immutability of Fields

Fields of frozen objects are not allowed to be modified. Hence, if a field of an object contains a reference to x at the time this object gets frozen, a prover should be able to assert anywhere later in the program execution that this field still holds the same reference to x. For that purpose, we follow the suggestion of Leino, Müller, and Wallenburg [7] and add a function called *UltimateValue*:

```
function UltimateValue<alpha>(o: ref, f: Field alpha) returns (alpha);
```

This function takes an object and a field of any reference or value type as inputs. The following axiom then states that for every frozen object, the value of all its fields can be determined solely through the return value of *UltimateValue*, i.e. only by looking at the object and the field, but not the heap:

```
axiom (∀<alpha> h: HeapType, o: ref, f: Field alpha • IsHeap(h) ∧ h[o, $frozen]
    ∧ f ≠ $frozen ∧ f ≠ $ownerRef ∧ f ≠ $ownerFrame ∧ ... ⇒
    h[o, f] == UltimateValue(o, f));
```

This means that no matter what the current heap is and, hence, no matter where in the program we are, the fields of a frozen object are always determined by the exact same function call. Therefore, the values of those fields stay the same forever, as a function can never return different values for the same input arguments. Or put differently: fields of a frozen object are determined independently from the heap.

We have to exclude some fields from this axiom that are internal to this axiomatization of the Spec# language. For clarity, we have only mentioned *$frozen*, *$ownerRef*, and *$ownerFrame* here. Not excluding these fields results in contradictions with the rest of the axiomatization, and a prover could then verify, for example, an expression like 'assert false'.

4.8 Triggers

To decrease the time a prover takes to verify programs, we may add triggers to the axioms, which give hints to the prover, with which values the axiom should be instantiated. For clarity, we have omitted these triggers in the axioms presented so far. Many of them are only important to improve the performance of static verification, but some are also crucial for the correctness of the

verification results, as a prover would sometimes not instantiate the axioms with the necessary values without these triggers and, therefore, not come to the correct conclusions.

Most triggers are straight forward and just repeat one or more expressions that occur in the axiom. As an example, we repeat here the axiom from the previous section, but this time including the triggers:

```
axiom (∀<alpha> h: HeapType, o: ref, f: Field alpha • { h[o, f] } { IsHeap(h),
    UltimateValue(o, f) } IsHeap(h) ∧ h[o, $frozen] ∧ f ≠ $frozen ∧
    f ≠ $ownerRef ∧ f ≠ $ownerFrame ∧ ... ⇒ h[o, f] == UltimateValue(o, f));
```

These two triggers simply tell a prover to only instantiate the axiom with those heaps, objects, and fields that are connected through the expression 'h[o, f]' or 'UltimateValue(o, f)'. We have to include the 'IsHeap(h)' expression in the second trigger, because each bound variable has to be mentioned in every trigger. Adding these triggers can reduce the work a prover has to do significantly.

Finding the best triggers for all axioms has not been easy, and there is also not one best solution. Sometimes adding a trigger can reduce the time Boogie takes to verify some test cases, while other test cases suddenly take much longer, or they cannot even be correctly verified anymore.

The triggers we have chosen are therefore a compromise, primarily ensuring the correctness of the verification results and secondarily trying to improve the average verification time for test cases as best as possible.

4.9 Prelude Class

The class *Prelude* provides the connection between the *Prelude* file and the Boogie source code. String constants are declared in this class for those functions and constants defined in the *Prelude* file that have to be accessed or called from the Boogie source code.

In order to use our new frozen objects axiomatization in Boogie, we therefore add four new string constants to the class. The string values have to match the names of the procedure and functions as defined in the *Prelude* file:

```
public const string! IsFrozen = "IsFrozen";
public const string! UpdateOwnersForFrozenProcName = "$UpdateOwnersForFrozen";
public const string! AsFrozenField = "AsFrozenField";
public const string! AsElementsFrozenField = "AsElementsFrozenField";
```

In the next chapter, we show how those string constants enable us to call this procedure and these functions from the source code of Boogie, helping us to translate Spec# programs using frozen objects into BoogiePL.

Chapter 5

Translation into BoogiePL

Boogie can be invoked either directly with a BoogiePL program or we can invoke the SscBoogie program with a compiled Spec# file. SscBoogie first translates the process (.exe) or library (.dll) assemblies into BoogiePL, before verification conditions can be generated.

In this chapter, we look at how the Spec# extensions for frozen objects are translated into BoogiePL. These translations may use the procedure and functions introduced in the previous chapter, as the content of *Prelude* is added to each BoogiePL program that has been generated from a Spec# source.

5.1 Freeze Keyword

As we have seen in Chapter 3.2, the Spec# compiler translates each occurrence of the *freeze* keyword into a method call to the static method *FreezeHelpers.Freeze*. On runtime, this method does nothing, but it should freeze the provided object during verification.

In method *HandleSpecialCaseMethods* of class *InstructionTranslator*, we therefore replace each call to method *FreezeHelpers.Freeze* with an implementation that takes care of freezing the provided object:

```
case "Microsoft.Contracts.FreezeHelpers.Freeze(
    optional(Microsoft.Contracts.NonNullType) System.Object)":
{
  sink.Comment(statement, "FreezeHelpers.Freeze");
  assert arguments.Count == 1;

  // get the object reference provided as argument to the method
  Bpl.Expr! subject = TranslateLocal((Cci.Variable!)arguments[0],
      statement, Role.Use);

  // assert subject.$ownerFrame == $PeerGroupPlaceholder;
  Bpl.Expr cond = om.HasNoOwner(subject, sink.HeapExpr());
  currentBlock.Cmds.Add(Sink.Assert(cond, statement,
      "freezing the subject requires it to be unowned"));

  // assert IsPeerConsistent(subject);
  cond = om.IsPeerConsistent(subject, false);
  currentBlock.Cmds.Add(Sink.Assert(cond, statement,
      "the subject must be peer consistent in order to be frozen"));

  // call $UpdateOwnersForFrozen(subject);
  Bpl.Cmd c = new Bpl.CallCmd(NoToken, Prelude.UpdateOwnersForFrozenProcName,
```

```
      new ExprSeq(subject), new IdentifierExprSeq());
  this.currentBlock.Cmds.Add(c);

  return true;
}
```

We start by translating the provided object reference into an expression we call *subject*. Then we ensure that *subject* is not owned already by looking at its owner frame. We check that it still contains the value *$PeerGroupPlaceholder*, meaning that no owner has been assigned to *subject* yet. We also need to ensure that *subject* is peer consistent, i.e. neither *subject* nor any of its peers may be exposed at the time the freezing takes place. Finally, we call the *$UpdateOwnersForFrozen* procedure from the *Prelude* file on *subject* by using the string constant *Prelude.UpdateOwnersForFrozenProcName* that we defined in Chapter 4.9. This procedure call then takes care of assigning ($freezerRef, $Freezer) to ($ownerRef, $ownerFrame) of *subject* as explained in Chapter 4.2.

In BoogiePL code, the translation of a statement 'freeze obj' therefore consists of two assertions, ensuring that the object is allowed to be frozen, and a procedure call that takes care of the freezing:

```
assert $Heap[obj, $ownerFrame] == $PeerGroupPlaceholder;
assert (∀ $pc: ref • $pc ≠ null ∧ $Heap[$pc, $allocated] ∧
    $Heap[$pc, $ownerRef] == $Heap[obj, $ownerRef] ∧
    $Heap[$pc, $ownerFrame] == $Heap[obj, $ownerFrame] ⇒
    $Heap[$pc, $inv] == $typeof($pc) ∧ $Heap[$pc, $localinv] == $typeof($pc));
call $UpdateOwnersForFrozen(obj);
```

5.2 IsFrozen Check

Every call to method *FreezeHelpers.IsFrozen*, which at runtime always returns *true*, gets replaced during the translation into BoogiePL by the following implementation given in method *TranslateCall* of class *ExpressionTranslator*:

```
case "Microsoft.Contracts.FreezeHelpers.IsFrozen$System.Object":
{
  assert operands.Count == 1;

  // get the object reference provided as argument to the method
  Bpl.Expr! subject = TranslateExpression((!) operands[0], heapName);

  // assert $IsFrozen(arg0, $Heap);
  return sink.IsFrozen(subject);
}
```

We first translate the provided object reference into an expression called *subject*. Then we call the helper method *IsFrozen* with *subject* as parameter, which takes care of calling the corresponding *IsFrozen* function in the *Prelude* file. During verification, *FreezeHelpers.IsFrozen* thus not always returns true anymore, only when the provided object is frozen. The translation into BoogiePL of the following Spec# code:

```
assert FreezeHelpers.IsFrozen(obj);
```

looks therefore simply like that:

```
assert IsFrozen(obj, $Heap);
```

where *IsFrozen* in the BoogiePL code does not refer to the method in class *FreezeHelpers*, but to the function of the same name defined in the *Prelude* file.

5.3 [Frozen] Modifier

For every field declared as [Frozen], we have to emit an axiom encoding, in terms of the axiomatization introduced in Chapter 4, that the object referred to by this field is frozen:

```
if (field.IsFrozen) {
  // for frozen fields, emit: axiom AsFrozenField(f) = f
  Bpl.Expr! ax = Bpl.Expr.Eq(Function(Sink.BuiltinFunction.AsFrozenField,
      Sink.IdentWithClean(fieldConst)), Sink.IdentWithClean(fieldConst));
  this.earlyDeclarations.Add(new Bpl.Axiom(NoToken, ax));
}
```

When we assign an object to a frozen field, then it is checked whether this object is either already frozen or valid to be frozen, before the assignment takes place. In method *VisitStoreField* of class *InstructionTranslator* we take care of this:

```
if (field.IsFrozen) {
  // assert source == null ∨ IsFrozen(source) ∨
  //     source.$ownerFrame == $PeerGroupPlaceholder;
  Bpl.Expr previousOwner = Bpl.Expr.Or(sink.IsFrozen(source),
      om.HasNoOwner(source, sink.HeapExpr()));
  if (!Sink.IsNonNullType(field.Type))
    previousOwner = Bpl.Expr.Or(Bpl.Expr.Eq(source, sink.Null), previousOwner);
  this.currentBlock.Cmds.Add(Sink.Assert(previousOwner, statement,
      "illegal assignment to frozen field, RHS may already be owned"));

  // assert source ≠ null ∨ source.$ownerFrame == $PeerGroupPlaceholder ⇒
  //     IsPeerValid(source);
  Bpl.Expr ownerChanges = om.HasNoOwner(source, sink.HeapExpr());
  if (!Sink.IsNonNullType(field.Type))
    ownerChanges = Bpl.Expr.And(Bpl.Expr.Neq(source, sink.Null), ownerChanges);
  Bpl.Expr cond = Bpl.Expr.Imp(ownerChanges, om.IsPeerValid(source));
  this.currentBlock.Cmds.Add(
      Sink.Assert(cond, statement, "RHS and its peers must all be valid"));

  // call $UpdateOwnersForFrozen(source);
  Bpl.Cmd c = new Bpl.CallCmd(NoToken, Prelude.UpdateOwnersForFrozenProcName,
      new ExprSeq(source), new IdentifierExprSeq());
  this.currentBlock.Cmds.Add(c);
}
```

The variable *source* holds the object we want to assign to the frozen field. First, we assert that *source* is either *null*, already frozen, or unowned. Then we assert that if the object is unowned, it has to be peer valid, because we are not allowed to freeze a currently exposed object. Finally, we call the *$UpdateOwnersForFrozen* procedure no matter whether we have *null*, a frozen object, or an unowned object as *source*. In the first two cases, the procedure ensures that nothing is changed, while in case of *source* being unowned, the procedure takes care of freezing the object.

A [Frozen] modifier attached to a method parameter or return value gets translated into a pre- or postcondition, stating that if the corresponding object is not *null*, it has to be frozen on entry or exit of the method respectively.

The following variation of the *WeatherReport* class exemplifies the three different uses of the [Frozen] modifier, namely attached to a field, a parameter, and a return value:

```
public class WeatherReport {
  [Frozen] private TempMeasurement temp;
```

```
  [return: Frozen]
  public TempMeasurement AddAndReturnTemp([Frozen] TempMeasurement! tm)
    modifies this.temp;
  {
    temp = tm;
    return temp;
  }
}
```

Translated into BoogiePL and reduced to the essential parts, mainly showing the code being emitted for the three [Frozen] modifiers, the above example looks as follows:

```
const unique WeatherReport.temp: Field ref;

axiom AsFrozenField(WeatherReport.temp) == WeatherReport.temp;

procedure WeatherReport.AddAndReturnTemp$TempMeasurement(this: ref, tm: ref)
    returns ($result: ref);
  ...
  // tm is frozen
  requires IsFrozen(tm, $Heap);
  ...
  // return value is frozen
  ensures $result == null ∨ IsFrozen($result, $Heap);
  ...

implementation WeatherReport.AddAndReturnTemp$TempMeasurement(this: ref, tm: ref)
    returns ($result: ref)
{
  ...
  assert tm == null ∨ IsFrozen(tm, $Heap) ∨
      $Heap[tm, $ownerFrame] == $PeerGroupPlaceholder;
  assert tm ≠ null ∧ $Heap[tm, $ownerFrame] == $PeerGroupPlaceholder ⇒
      (∀ $pc: ref • $pc ≠ null ∧ $Heap[$pc, $allocated] ∧
      $Heap[$pc, $ownerRef] == $Heap[tm, $ownerRef] ∧
      $Heap[$pc, $ownerFrame] == $Heap[tm, $ownerFrame] ⇒
      $Heap[$pc, $inv] == $typeof($pc) ∧ $Heap[$pc, $localinv] == $typeof($pc));
  call $UpdateOwnersForFrozen(tm);
  ...
  $Heap[this, WeatherReport.temp] := tm;
  ...
  $result := $Heap[this, WeatherReport.temp];
  return;
}
```

The field *WeatherReport.temp* is defined as a constant of type *Field ref* and declared as frozen with the help of function *AsFrozenField*. Procedure *AddAndReturnTemp* has a precondition, requiring that parameter *tm* has to be frozen. An additional check 'tm == null' is not necessary, because *tm* is declared as a non-null type. The procedure also has a postcondition, ensuring that the result is either *null* or frozen.

In the implementation, we see that the two axioms and the call to *$UpdateOwnersForFrozen* are placed before the actual assignment of *tm* to field *WeatherReport.temp*. This same field is then assigned to variable *$result*, guaranteeing the return value to be frozen as expected by the corresponding postcondition.

5.4 [ElementsFrozen] Modifier

For fields declared with the [ElementsFrozen] modifier, we cannot just emit one axiom as with [Frozen]. Because [ElementsFrozen] may apply only to some elements, we first have to retrieve all element positions that should be frozen.

We can call the new static method *FieldElementsFrozenPositions* for that purpose, which takes a field as input and returns a list of all element positions of this field that should be frozen:

```
public static List<int> FieldElementsFrozenPositions(Field f) {
  List<int> res = new List<int>();

  if (f == null) return res;

  TypeNode type = TypeNode.DeepStripModifiers(f.Type);

  // elements of an array of an immutable type are frozen
  if (type is ArrayType) {
    TypeNode t = ((ArrayType)type).ElementType;
    if (!t.IsValueType && TypeNode.IsImmutable(t))
      res.Add(-1);
  }

  // in a generic class type arguments of an immutable type are frozen
  if (type.TemplateArguments != null) {
    for (int i = 0; i < type.TemplateArguments.Count; i++) {
      TypeNode t = TypeNode.StripModifiers(type.TemplateArguments[i]);
      if (!t.IsValueType && TypeNode.IsImmutable(t))
        res.Add(i);
    }
  }

  // elements of fields marked with the [ElementsFrozen] modifier are frozen
  AttributeList al = f.GetAllAttributes(SystemTypes.ElementsFrozenAttribute);
  if (al == null) return res;
  foreach (AttributeNode attr in al) {
    ExpressionList exprs = attr.Expressions;
    if (exprs == null || exprs.Count == 0) {
      if (type is ArrayType) {
        res.Add(-1);
      } else if (type.IsGeneric || type.Template != null) {
        TypeNodeList! tnl = (!)type.TemplateArguments;
        for (int i = 0; i < tnl.Count; i++) {
          if (tnl[i].IsReferenceType) res.Add(i);
        }
      }
    } else {
      Expression arg = exprs[0];
      Literal lit = arg as Literal;
      if (lit != null && lit.Value is int) res.Add((int)lit.Value);
    }
  }

  return res;
}
```

When the provided *Field* object *f* is not *null*, we first check whether the field is an array of an immutable type. The elements of such an array are always frozen, and we therefore add -1 to the list. Next, if the field is of a generic type, we look for type parameters of an immutable type and add the respective positions to the list. Finally, we check for the [ElementsFrozen] modifier and distinguish between a modifier with or without an associated value. If the modifier has no value, then we add -1 to the list if we have an array or, in case of a generic type, add to the list all positions of the type parameters, i.e. we would add {0,1,2} for a generic type with three type parameters. If the modifier has a value associated with it, then we just add this value to the list of frozen elements.

For each position returned by this method, we then emit an axiom using this position as argument to the *AsElementsFrozenField* function:

```
foreach (int pos in (!)Util.FieldElementsFrozenPositions(field)) {
  // for ElementsFrozen fields, emit: axiom AsElementsFrozenField(f, pos) = f
  Bpl.Expr! ax = Bpl.Expr.Eq(Function(Sink.BuiltinFunction.AsElementsFrozenField,
      Sink.IdentWithClean(fieldConst), Bpl.Expr.Literal(pos)),
      Sink.IdentWithClean(fieldConst));
  this.earlyDeclarations.Add(new Bpl.Axiom(NoToken, ax));
}
```

When we assign an object to a field specified as [ElementsFrozen], we once again retrieve all element positions that should be frozen. For each position, we then assert that if the *source* object is not *null*, the *$ElementProxy* object at the given position has to be either frozen or unowned. After the assertion, we call *$UpdateOwnersForFrozen* with the element proxy at this position as parameter.

Attached to a parameter or a return value, [ElementsFrozen] gets translated into one or more pre- or postconditions, which require or ensure the specified elements to be either *null* or frozen.

The following example shows how to use the [ElementsFrozen] modifier in three different scenarios. As a modifier to the field *dictionary*, [ElementsFrozen] declares both type parameters of type *Foo* as frozen. The method *Bar* requires both type parameters of *dic* to be frozen and ensures that the first type parameter of the return value is frozen:

```
public class Foo {
  [ElementsFrozen] Dictionary<Foo, Foo> dictionary;

  [return: ElementsFrozen(0)]
  public Dictionary<Foo, Foo> Bar([ElementsFrozen] Dictionary<Foo, Foo>! dic) {
    dictionary = dic;
    return dictionary;
  }
}
```

The translation into BoogiePL, of those parts of this example that are directly related to any of the three [ElementsFrozen] modifiers, is given below:

```
const unique Foo.dictionary: Field ref;

axiom AsElementsFrozenField(Foo.dictionary, 0) == Foo.dictionary;
axiom AsElementsFrozenField(Foo.dictionary, 1) == Foo.dictionary;

procedure Foo.Bar$System.Collections.Generic.Dictionary‘2...Foo‘~‘Foo(
    this: ref, dic: ref) returns ($result: ref);
  ...
  // elements of dic are frozen
  requires IsFrozen($ElementProxy(dic, 0), $Heap);
  requires IsFrozen($ElementProxy(dic, 1), $Heap);
```

```
  ...
  // elements of result are frozen
  ensures $result == null ∨ IsFrozen($ElementProxy($result, 0), $Heap);
  ...

implementation Foo.Bar$System.Collections.Generic.Dictionary‘2...Foo‘~‘Foo(
    this: ref, dic: ref) returns ($result: ref)
{
  ...
  assert $Heap[$ElementProxy(dic, 0), $ownerFrame] == $PeerGroupPlaceholder ∨
      dic == null ∨ IsFrozen($ElementProxy(dic, 0), $Heap);
  call $UpdateOwnersForFrozen($ElementProxy(dic, 0));
  assert $Heap[$ElementProxy(dic, 1), $ownerFrame] == $PeerGroupPlaceholder ∨
      dic == null ∨ IsFrozen($ElementProxy(dic, 1), $Heap);
  call $UpdateOwnersForFrozen($ElementProxy(dic, 1));
  ...
  $Heap[this, Foo.dictionary] := dic;
  ...
  $result := $Heap[this, Foo.dictionary];
  return;
}
```

The field *Foo.dictionary* is defined as a constant of type *Field ref*, and its two type parameters are declared as frozen by using function *AsElementsFrozenField*. Procedure *Bar* requires the element proxy of parameter *dic* to be frozen at positions 0 and 1 and ensures that if the return value is not *null*, the element proxy of this return value is frozen at least for position 0. Obviously, the element proxy is also frozen at position 1 in this particular example, but we just wanted to demonstrate the difference between using the [ElementsFrozen] modifier with or without an associated value.

In the implementation of method *Bar*, the procedure calls to *$UpdateOwnersForFrozen* together with their prefixed assertion appear consecutively for both type parameters, followed by the assignment of parameter *dic* to field *Foo.dictionary*. Finally, this field gets assigned to the *$result* variable and, thereby, ensures that the first type parameter is frozen as expected by the corresponding postcondition.

5.5 [ElementsCaptured] Modifier

Translating a Spec# program that uses the [ElementsCaptured] modifier into BoogiePL can be split into two separate tasks. On the one hand, we need to ensure that the element proxy of every parameter declared as [ElementsCaptured] - or the receiver object if the modifier is applied to the method - is unowned at either all or just the given positions on entry of a method. On the other hand, we also have to extend the frame condition of the method. We need to allow those *$ownerRef* and *$ownerFrame* references to be modified by the method, which belong to an element targeted by the [ElementsCaptured] modifier or to a peer of at least one of those elements.

Adding the correct preconditions to a method with a parameter specified as [ElementsCaptured] is not trivial, because we have to distinguish three different cases. The following code from method *MethodSignature* takes care of adding those preconditions, requiring the element proxy of the parameter to be unowned at the given positions:

```
foreach (Cci.AttributeNode! attr in (!)parameter.GetAllAttributes(
    Cci.SystemTypes.ElementsCapturedAttribute)) {
  Cci.ExpressionList exprs = attr.Expressions;
  if (exprs != null && exprs.Count != 0) {
    Cci.Expression arg = exprs[0];
```

```
    Cci.Literal lit = arg as Cci.Literal;
    if (lit != null && lit.Value is int)
      AddPreconditionForElementsCaptured(param, (int)lit.Value, name, isNonNull);
  } else {
    if (parameter.Type.IsGeneric || parameter.Type.Template != null) {
      if (parameter.Type.TemplateArguments != null) {
        for (int j = 0; j < parameter.Type.TemplateArguments.Count; j++) {
          if (parameter.Type.TemplateArguments[j].IsReferenceType)
            AddPreconditionForElementsCaptured(param, j, name, isNonNull);
        }
      }
    } else
      AddPreconditionForElementsCaptured(param, -1, name, isNonNull);
  }
}
```

For each parameter with an [ElementsCaptured] modifier, we first distinguish whether the modifier has an associated value. If it has one, we call the method *AddPreconditionForElementsCaptured* with the value as parameter. This method adds a precondition, requiring the element proxy of the parameter to be unowned at the provided position. Otherwise, we have to distinguish between an array and a generic type. For generic types, we require the element proxy to be unowned at all positions of type parameters that are of a reference type. For arrays, we require the *$ElementProxy* object to be unowned at position -1.

Method *GenerateObjectsFrameCondition* is responsible for adding the frame condition. We have to extend this method to also take parameters declared as [ElementsCaptured] into account when generating the frame condition. Otherwise, we would always get a frame condition violation, whenever an owner is assigned to the element proxy of such a parameter inside the method, because by default it is not allowed to modify the *$ownerRef* and *$ownerFrame* references of objects not created in the method itself.

For each parameter specified as [ElementsCaptured], we therefore have to add an expression to the frame condition, allowing the *$ownerRef* and *$ownerFrame* reference of the element proxy at the captured position to be modified inside the method. As assigning an owner to an object additionally affects all the peers of that object, we also have to exclude the owners of all peers of the captured element from having to remain unchanged.

The following example has a method *Bar* that assigns its parameter to the field *dic*. When the first type parameter of *dic* is not already frozen before the call to *Bar*, the method parameter needs to be declared as [ElementsCaptured(0)] for the assignment to be valid:

```
public class Test {
  [ElementsFrozen(0)] Dictionary<object, string> dic;

  public void Bar([ElementsCaptured(0)] Dictionary<object, string>! dic) {
    this.dic = dic;
  }
}
```

The following BoogiePL code is an excerpt of this example's translation, focusing on the [ElementsCaptured] modifier:

```
procedure Test.Bar$System.Collections.Generic.Dictionary‘2...System.Object‘~‘
    System.String(this: ref, dic: ref);
  ...
  requires (dic == null ∨ $Heap[$ElementProxy(dic, 0), $ownerFrame] ==
      $PeerGroupPlaceholder);
  ...
```

```
// frame condition
ensures (∀<alpha> $o: ref, $f: Field alpha • IncludeInMainFrameCondition($f) ∧
    $o ≠ null ∧ ... ∧ ¬(($f == $ownerRef ∨ $f == $ownerFrame) ∧
    old($Heap[$o, $ownerRef] == $Heap[$ElementProxy(dic, 0), $ownerRef] ∧
    $Heap[$o, $ownerFrame] == $Heap[$ElementProxy(dic, 0), $ownerFrame])) ∧
    ... ⇒ old($Heap)[$o, $f] == $Heap[$o, $f]);
...
```

The procedure *Bar* requires parameter *dic* to be either *null* or else its element proxy has to be unowned at position 0. The frame condition is an implication, ensuring that fields of all objects remain unmodified, except for those explicitly excluded in the antecedent. We truncated this antecedent for clarity, as normally it would be much longer. The part added because of the [ElementsCaptured] modifier states that if field f is either equal to *$ownerRef* or *$ownerFrame*, and object o has the same owner as the element proxy of *dic* at position 0, then f is allowed to be modified inside method *Bar*.

Chapter 6

Immutable Classes and Strings

The concept of immutable classes can be viewed as a subset of the frozen objects methodology, in which either no object of a class gets ever frozen, or all objects of a class get frozen right after construction. This implies that we may exchange the current implementation of immutable classes and, instead, use frozen objects in the background also to implement immutable classes. We therefore can have both, immutable classes and frozen objects, without the need for two different implementations to declare objects as immutable.

6.1 Removing Immutable Classes Axiomatization

We first remove the whole previous axiomatization for immutable classes, that has been discussed shortly in Chapter 2.3.3, from the *Prelude* file. This includes removing all the functions and axioms that have been written specifically to support the concept of immutable classes.

In addition, the following axiom has so far used the *$IsImmutable* function. It expresses that the elements of an array are either *null*, of an immutable type, or have the same owner as the element proxy of the array at position -1:

```
axiom (∀ a: ref, i: int, heap: HeapType • IsHeap(heap) ∧
    $typeof(a) <: System.Array ⇒ ArrayGet(heap[a, $elementsRef], i) == null ∨
    $IsImmutable($typeof(ArrayGet(heap[a, $elementsRef], i))) ∨
    (heap[ArrayGet(heap[a, $elementsRef], i), $ownerRef] ==
    heap[$ElementProxy(a, -1), $ownerRef] ∧
    heap[ArrayGet(heap[a, $elementsRef], i), $ownerFrame] ==
    heap[$ElementProxy(a, -1), $ownerFrame]));
```

But as the *$IsImmutable* function has now been removed from the *Prelude* file, the following part of the axiom, stating that the array elements are immutable, has to be removed as well:

```
$IsImmutable($typeof(ArrayGet(heap[a, $elementsRef], i)))
```

As we discuss in the next section, objects of an immutable type are also owned by the *Freezer* in the new implementation. The third part of the axiom, requiring the elements to have the same owner as the *$ElementProxy* object, therefore, already covers those objects, and we do not need to add anything to the axiom as a substitution for the removed expression. The reduced axiom now simply states that the elements of an array are either *null* or have the same owner as the element proxy of the array.

We also have to remove the following constants from the *Prelude* class, as they are no longer available, because they belonged to the old axiomatization of immutable classes:

```
public const string! IsImmutableFunction = "$IsImmutable";
public const string! AsImmutableFunction = "$AsImmutable";
```

```
public const string! AsMutableFunction = "$AsMutable";
```

This further requires us to remove all source code parts of Boogie that have used any of the above constants. Essentially, this leads to the complete removal of the old implementation of immutable classes from the Boogie verifier.

6.2 New Implementation for Immutable Classes

After removing the previous axiomatization of immutable classes, we now have to implement the handling of immutable types in a way that they depend on the axiomatization for frozen objects instead.

We start by treating fields of an immutable type as though they were specified with the [Frozen] modifier, i.e. for all such fields we have to emit an axiom declaring this field as frozen:

```
if (IsImmutable(field.Type, out isImmutable) && isImmutable) {
  // for fields of an immutable type, emit: axiom AsFrozenField(f) = f
  Bpl.Expr! ax = Bpl.Expr.Eq(Function(Sink.BuiltinFunction.AsFrozenField,
      Sink.IdentWithClean(fieldConst)), Sink.IdentWithClean(fieldConst));
  this.earlyDeclarations.Add(new Bpl.Axiom(NoToken, ax));
}
```

The method *IsImmutable*, which is not to be confused with function *$IsImmutable* that we removed from the *Prelude* file in the previous section, returns *true*, whenever it can be decided whether the provided field type is immutable. The out parameter *isImmutable* then holds the boolean value stating if the type is immutable or not. The *IsImmutable* method returns *false*, when the provided type is either *object*, as this is the only type allowed to have mutable as well as immutable subclasses, or if it is an interface type.

To ensure that an object of an immutable type is not modifiable anymore after its constructor has terminated, we freeze each object of such a type at the end of its initialization. For that purpose, we add the following code to method *VisitCall* of class *InstructionTranslator*:

```
// freeze object if it is of an immutable type
if (receiver != null && receiver.NodeType != Cci.NodeType.This &&
    callee.NodeType == Cci.NodeType.InstanceInitializer &&
    !(methodSignature.method.NodeType == Cci.NodeType.InstanceInitializer &&
    methodSignature.method.DeclaringType == callee.DeclaringType) &&
    type != null && Sink.IsImmutable(type, out isImmutable) && isImmutable) {
  Bpl.IdentifierExpr recExpr = this.TranslateLocal(receiver, sink.Convert(type));

  // assert recExpr.$ownerFrame == $PeerGroupPlaceholder;
  Bpl.Expr cond = om.HasNoOwner(recExpr, sink.HeapExpr());
  currentBlock.Cmds.Add(Sink.Assert(cond, statement,
      "freezing the subject requires it to be unowned"));

  // assert IsPeerConsistent(recExpr);
  cond = om.IsPeerConsistent(recExpr, false);
  currentBlock.Cmds.Add(Sink.Assert(cond, statement,
      "the subject must be peer consistent in order to be frozen"));

  // call $UpdateOwnersForFrozen(recExpr);
  Bpl.Cmd c = new Bpl.CallCmd(NoToken, Prelude.UpdateOwnersForFrozenProcName,
      new ExprSeq(recExpr), new IdentifierExprSeq());
  currentBlock.Cmds.Add(c);
}
```

Basically, whenever we have a method call that is a constructor (called *InstanceInitializer* in the source code), and the declaring type of this constructor is immutable, we first assert that the receiver object is unowned and peer consistent, and then call the *$UpdateOwnersForFrozen* procedure on this receiver.

There are, however, some special cases that we have to exclude. When the constructor of a class calls its superclass constructor, then we do not want the newly created object to be frozen at the end of this 'base()' call, because we would get an error at the end of the subclass constructor, when the attempt to freeze the object for a second time fails, as no object may be frozen twice. The expression 'receiver.NodeType != Cci.NodeType.This' in the if-clause of the above code, therefore, prevents an object to be frozen at the end of a constructor of an immutable class, if the receiver object is the *this* instance as is the case with a 'base()' call.

Another problem arises, when an immutable class has two different constructors and one of them is calling the other. In such a case, we also do not want the object to get frozen after the call of one constructor to the other returns. The expression:

```
!(methodSignature.method.NodeType == Cci.NodeType.InstanceInitializer &&
    methodSignature.method.DeclaringType == callee.DeclaringType)
```

in the if-clause of the above code ensures that if the method calling the constructor is also a constructor, and this calling method has the same declaring type (i.e. both methods are in the same class), then we do not freeze the object at the end of such a constructor call.

The class *Test* in the following example contains a field of the immutable type *Foo* and a method *Bar* that assigns a new *Foo* object to this field:

```
[Immutable]
public class Foo { }

public class Test {
  public Foo foo;

  public void Bar() {
    foo = new Foo();
  }
}
```

The translation of this example into BoogiePL, limited to the parts that show the new implementation behind the concept of immutable classes, looks as follows:

```
const unique Test.foo: Field ref;

axiom AsFrozenField(Test.foo) == Test.foo;

implementation Test.Bar(this: ref)
{
  ...
  call Foo..ctor(stack50000o);
  assert $Heap[stack50000o, $ownerFrame] == $PeerGroupPlaceholder;
  assert (∀ $pc: ref • $pc ≠ null ∧ $Heap[$pc, $allocated] ∧
      $Heap[$pc, $ownerRef] == $Heap[stack50000o, $ownerRef] ∧
      $Heap[$pc, $ownerFrame] == $Heap[stack50000o, $ownerFrame] ⇒
      $Heap[$pc, $inv] == $typeof($pc) ∧ $Heap[$pc, $localinv] == $typeof($pc));
  call $UpdateOwnersForFrozen(stack50000o);
  ...
  $Heap[this, Test.foo] := stack50000o;
  ...
}
```

The function *AsFrozenField* declares the field *Test.foo* as frozen. Hence, this field is indistinguishable in the BoogiePL code from a field of a mutable type specified as [Frozen]. In the implementation of *Bar*, the constructor of immutable class *Foo* is called on the newly allocated variable *stack50000o*. Then it is checked with two assertions whether this variable is unowned and peer consistent. Finally, *stack50000o* gets frozen through the function call to *$UpdateOwnersForFrozen*, and only then is it assigned to field *Test.foo*.

Array fields of an immutable element type and generic fields with one or more immutable type parameters are handled similar than fields specified as [ElementsFrozen]. We have to emit the corresponding *AsElementsFrozenField* axioms and freeze the *$ElementProxy* object of such fields at the correct position after the end of the constructor.

Method parameters and return values of an immutable type get treated as though specified as [Frozen] by adding the corresponding pre- or postcondition to the method, requiring such parameters to be either *null* or frozen and ensuring that a return value of an immutable type is *null* or frozen.

We also need to declare local variables of an immutable type as frozen. For that purpose, we extend the where-clause of those local variables with an implication, stating that if the variable does not contain *null*, it has to be frozen. Without this additional clause, the assertion in the following example could not be verified:

```
[Immutable]
public class Foo { }

public class Test {
  public void Bar() {
    Foo foo = new Foo();

    for(int i = 0; i < 4; i++) {
      if (foo != null)
        assert FreezeHelpers.IsFrozen(foo);
      foo = new Foo();
    }
  }
}
```

The assertion in this loop should always hold, because the object referenced by *foo* is frozen at the entry of the loop, and the loop body preserves this fact despite assigning yet another *Foo* object to variable *foo* in each loop iteration. A prover can, however, get confused by these new assignments and might not be able to verify this assertion anymore, which is why the help of the additional where-clause is required.

In the BoogiePL translation, the declaration of local variable *foo* with the extended where-clause looks as follows:

```
var foo: ref where $Is(foo, Foo) ∧ $Heap[foo, $allocated] ∧ (foo ≠ null ⇒
    IsFrozen(foo, $Heap));
```

With all those modifications, the implementation of immutable classes now completely relies on the frozen objects methodology. Before, objects of an immutable type had to be treated separately in many parts of the Boogie source code, whereas now, those objects get simply frozen and can then be treated the same as every other owned object. The unexposable *Freezer* as owner of those objects prevents any modification of them without the need of special checks.

Thus, with this new implementation for immutable classes, we not only reduce the size of the Boogie source code but also the size of BoogiePL files by not needing all the special checks for immutability anymore, as there is no need to distinguish between immutable and mutable objects. Violations of the immutability properties now get detected by the same assertions that are already there to check for the correct use of ownership and, therefore, add no additional overhead.

6.3 Precondition for Generic Parameters

In a generic class, if a method's parameter has a generic type parameter of this class as its type, it is required that the object provided to this method has the same owner as the element proxy of the generic type at the position representing the corresponding type parameter.

In class *Foo* of the following example, we have a field *fooList* that is a generic list of *Foo* elements. In method *Bar*, we then want to add the parameter *foo* to this list:

```
public class Foo {
  [Peer] [ElementsPeer] public List<Foo>! fooList = new List<Foo>();

  public void Bar(Foo! foo)
    modifies fooList.*;
  {
    fooList.Add(foo);
  }
}
```

Translated into BoogiePL, method *Add* previously had the following precondition, requiring item *foo* to be either *null*, be of an immutable type, or have the same owner as the element proxy of *fooList* at position 0:

```
// item is a peer of the expected elements of the generic object
requires item == null ∨ $IsImmutable($typeof(item) ∨
   ($Heap[item, $ownerRef] == $Heap[$ElementProxy(this, 0), $ownerRef] ∧
    $Heap[item, $ownerFrame] == $Heap[$ElementProxy(this, 0), $ownerFrame]);
```

With the removal of the immutability axiomatization, we may no longer use the second part of this precondition, as the function *$IsImmutable* no longer exists. Hence, we have to reduce the precondition as follows:

```
// item is a peer of the expected elements of the generic object
requires item == null ∨
   ($Heap[item, $ownerRef] == $Heap[$ElementProxy(this, 0), $ownerRef] ∧
    $Heap[item, $ownerFrame] == $Heap[$ElementProxy(this, 0), $ownerFrame]);
```

Because *fooList* is specified as [ElementsPeer], the verification of this example fails, as the parameter *foo* is neither *null* nor a peer of *Foo*. By adding the precondition:

```
requires Owner.Same(fooList, foo);
```

to method *Bar*, we can solve this problem, because parameter *foo* is then guaranteed to be a peer of *Foo* same as the element proxy of *fooList* at position 0. The verification now succeeds. However, if we instead make *Foo* an immutable class and use it as the generic type parameter of the list, we run into another problem:

```
public class Test {
  [Peer] public List<Foo>! fooList = new List<Foo>();

  public void Bar(Foo! foo)
    modifies fooList.*;
  {
    fooList.Add(foo);
  }
}

[Immutable]
public class Foo { }
```

In the new implementation, both the type parameter of *fooList* and the object *foo* are frozen. But as one could be directly owned by the *Freezer* and the other just indirectly, they do not necessarily have the same owner. We therefore thought of replacing the *$IsImmutable* function in this precondition with the new *IsFrozen* function:

```
// item is a peer of the expected elements of the generic object
requires item == null ∨ IsFrozen(item, $Heap)
    ($Heap[item, $ownerRef] == $Heap[$ElementProxy(this, 0), $ownerRef] ∧
    $Heap[item, $ownerFrame] == $Heap[$ElementProxy(this, 0), $ownerFrame]);
```

This, however, turned out not to be a good idea. In the body of a method with this precondition, a prover cannot know whether *item* is frozen or has the same owner as the corresponding type parameter of the class, because only one has to hold, which restricts how this parameter may be used inside the method. Before, with the *$IsImmutable* function, we did not have this problem, as a prover could statically verify at each point in a program, if the type of a variable is immutable. Hence, a prover could always decide whether a parameter is of an immutable type or, in the other case, has the same owner as the corresponding type parameter.

The solution we then found, is not to add the *IsFrozen* function to the precondition, but instead to only emit this precondition, when the type of the parameter is not immutable. If it is of an immutable type, then we already emit a separate axiom, requiring the parameter to be frozen as discussed in the previous section.

The precondition still imposes one limitation for which we have not found a satisfying solution. If we, instead of an immutable type, use a mutable type as type parameter of our *fooList*, but specify it as [ElementsFrozen], then we cannot satisfy the precondition of the *Add* method:

```
public class Foo {
  [Peer] [ElementsFrozen] public List<Foo>! fooList = new List<Foo>();

  public void Bar([Frozen] Foo! foo)
    modifies fooList.*;
  {
    fooList.Add(foo);
  }
}
```

Not even specifying parameter *foo* of method *Bar* as [Frozen] helps us out here, because as argued above, two frozen objects do not necessarily have the same owner. And as we may provide mutable and frozen objects as parameter to *Add*, it is also no option to not emit the precondition at all as is possible for immutable types. Hence, using the [ElementsFrozen] modifier on *fooList* in this example severely restricts, what we may do with this list.

6.4 Strings

The translation of strings into BoogiePL is handled differently than that of all other objects. A string is not created by calling a constructor but declared as a constant. The properties of such a string constant are then described by an axiom:

```
const unique $stringLiteral2: ref;

axiom $IsNotNull($stringLiteral2, System.String) ∧
    $StringLength($stringLiteral2) == 4 ∧
    (∀ heap: HeapType • IsHeap(heap) ⇒ heap[$stringLiteral2, $allocated]) ∧ ...;
```

The constant *$stringLiteral2* is defined as being non-null, having a length of 4, and being allocated in every heap.

Because the type *String* is specified as [Immutable], we have to guarantee that every string instance is frozen. The assertion in the following example should therefore always hold:

```
public class Foo {
    public void Bar() {
        assert FreezeHelpers.IsFrozen("test");
    }
}
```

However, the above assertion, with which we want to ensure that a new string is frozen, cannot be verified without any further modifications. As no constructor is called to create a string, the source code where we freeze any object of an immutable type, as described in the previous section, is also not reached. We therefore have to explicitly state as a property of each string that it is frozen. We do so by extending the above axiom with an additional clause:

```
const unique $stringLiteral2: ref;

axiom ... ∧ (∀ heap: HeapType • IsHeap(heap) ⇒ IsFrozen($stringLiteral2, heap));
```

Now, Boogie is able to verify at each point in a program that a string is frozen, even if this string has not been assigned to any field or local variable as in the example above.

Chapter 7

Conclusion

Immutability of objects has been supported by Spec# through the concept of immutable classes. This concept does, however, not support the more fine-grained solution of declaring individual objects as immutable. The frozen objects methodology proposes a better solution, where any object is allowed to be frozen. In this thesis, we took this theoretical concept and implemented it into Spec#.

We had to adapt the concept to work with the specific properties of Spec#. Allowing no ownership transfer, we cannot freeze every mutable object in Spec# but only those that are unowned yet, and because not every field of an object has to be declared as [Peer], [Rep], or [Frozen] in Spec#, not every object reachable from a frozen object is guaranteed to be immutable.

We also exchanged the previous implementation for immutable classes with a new one that makes use of the newly implemented frozen objects methodology. This allowed us to reduce the amount of checks necessary to verify a program. The previous implementation added its own checks during verification, even if a program contained no immutable class, while the frozen objects methodology uses the checks that are added by the ownership model anyway and, hence, creates no additional overhead.

7.1 Testing and Open Source Integration

We have thoroughly tested this new implementation with a large test suite. We have found and fixed many bugs thanks to this test suite. The errors have mostly been found in those parts of the code that we modified, but some of those bugs have already been present in the previous implementation and might just not have had an impact without the use of frozen objects. In appendix C, we give a short explanation of those bugs and discuss, how we have fixed them.

We also created a subversion patch with our modifications of Spec# and Boogie for the open source version of the language [9]. Before doing so, we run our new implementation against all the test cases provided with the open source code. Most of those test cases could be verified the same as before, but some could not be verified anymore or have returned different error messages. Some problems have been caused by bugs in our implementation that we have subsequently fixed. The other problems, however, are due to the differences in the implementation, especially that objects of an immutable type are now owned by the *Freezer*, whereas they have been unowned before.

We have found three kinds of differences: we may get different error messages for the same errors, we may get new error messages, or we may not get some error messages anymore. In appendix B, we explain those differences and the reasons behind them in more detail.

7.2 Limitations

The frozen objects technique has only been implemented for the Boogie methodology and does not support the visible-state methodology.

As Boogie does not support static fields in combination with Spec#'s ownership model yet, and the frozen objects methodology relies on this ownership model's checks, we also have no support for static fields in the implementation of frozen objects.

For performance reasons, there are no runtime checks for the ownership model of Spec#, and as the error detection for frozen objects relies on these checks, we thus have no runtime checks for frozen objects either.

7.3 Experience

Working on this thesis has been a really interesting experience. Working with a large, not always very well documented code base has been challenging at first. The easier part has been to extend the Spec# compiler with the new *freeze* keyword and all the new modifiers that we have introduced. Besides declaring them, we only had to add some static compiler checks, for example, to detect invalid modifier combinations.

The most challenging part has definitely been the axiomatization of the frozen objects methodology. Even though most axioms have already been presented in the paper introducing the concept [7], writing them in a way to work correctly with all the features of the full Spec# language has been very difficult. Sometimes an axiom has been too restrictive, leading to test cases that could not be verified correctly, and sometimes an axiom has not been restrictive enough, leading to contradictions in the axiomatization.

A bug in the source code can be found by debugging the respective code. However, a contradiction in the axiomatization can only be detected by first removing one axiom after the other until the problem causing axiom has been found, and then trying to figure out what the problem with this axiom is, i.e. with which other axiom the contradiction can be obtained, and how we can fix this problem.

Leino, Müller, and Wallenburg [7] have used an example with travelers that share the same frozen map in their paper to illustrate the concept of frozen objects. We have written our own traveler example using the full Spec# language (see Appendix A.2) to demonstrate the capabilities of our implementation.

In the end, despite the above mentioned limitations of the frozen objects methodology implementation into Spec#, we are very happy with the achieved result, and we hope that this implementation finds its way into the open source version of the language.

Acknowledgements
I would like to thank my supervisors Joseph N. Ruskiewicz and Prof. Peter Müller for the help and support they have given me during the writing of this thesis.

Appendix A

Examples

A.1 Weather Report

This section contains the source code of the running *WeatherReport* example, which has been used several times in this paper to illustrate the benefits of the frozen objects technique. We first give the version using only immutable classes and then also provide the version making use of the frozen objects methodology.

A.1.1 Version with Immutable Classes

```
public class MainClass {
  public void Main() {
    Date date = new Date();
    Location loc = new Location();
    Temp airT = new Temp();
    Temp waterT = new Temp();
    Velocity windVel = new Velocity();

    // create a weather report
    WeatherReport wr = new WeatherReport(loc);

    // create a temperature and a wind measurement and
    // add them to the weather report
    wr.AddTemp(new TempMeasurement(date, loc, airT, waterT));
    wr.AddWind(new WindMeasurement(date, loc, windVel));
  }
}

public class WeatherReport {
  private TempMeasurement temp;
  private WindMeasurement wind;
  public Location loc;

  invariant temp == null || loc == temp.loc;
  invariant wind == null || loc == wind.loc;

  public WeatherReport(Location loc)
    ensures this.loc == loc;
  { this.loc = loc; }
```

```
  public void AddTemp(TempMeasurement! tm)
    requires loc == tm.loc;
    ensures loc == temp.loc;
    modifies this.temp;
  { temp = tm; }

  public void AddWind(WindMeasurement! wm)
    requires loc == wm.loc;
    ensures loc == wind.loc;
    modifies this.wind;
  { wind = wm; }

  public TempMeasurement GetTemp() {
    return temp;
  }

  public WindMeasurement GetWind() {
    return wind;
  }
}

[Immutable]
public class Measurement {
  public Date date;
  public Location loc;

  public Measurement(Date d, Location l)
    ensures date == d && loc == l;
  {
    date = d;
    loc = l;
  }
}

[Immutable]
public class TempMeasurement : Measurement {
  public Temp airTemp;
  public Temp waterTemp;

  public TempMeasurement(Date d, Location l, Temp airT, Temp waterT)
    ensures date == d && loc == l && airTemp == airT && waterTemp == waterT;
  {
    airTemp = airT;
    waterTemp = waterT;
    base(d, l);
  }
}

[Immutable]
public class WindMeasurement : Measurement {
  public Velocity windVelocity;

  public WindMeasurement(Date d, Location l, Velocity windVel)
    ensures date == d && loc == l && windVelocity == windVel;
```

```
  {
    windVelocity = windVel;
    base(d, l);
  }
}
```

A.1.2 Version with Frozen Objects

```
public class MainClass {
  public void Main() {
    Date date = new Date();
    Location loc = new Location();
    Temp airT = new Temp();
    Temp waterT = new Temp();
    Velocity windVel = new Velocity();

    // create a CurrentWeather object and
    // update the temperatures and the wind velocity
    CurrentWeather cr = new CurrentWeather(date, loc);
    cr.UpdateTemp(date, airT, waterT);
    cr.UpdateWind(date, windVel);

    // create a weather report
    WeatherReport wr = new WeatherReport(loc);

    // create a temperature measurement and add it to the weather report
    // correcting the measurement and freezing have to be done before the
    // call to AddTemp because this method already expects a frozen object
    TempMeasurement tm = new TempMeasurement(date, loc, airT, waterT);
    tm.CorrectMeasurement();
    freeze tm;
    wr.AddTemp(tm);

    // create a wind measurement and add it to the weather report
    // correcting the measurement and freezing takes place during
    // the AddWind method call
    WindMeasurement wm = new WindMeasurement(date, loc, windVel);
    wr.AddWind(wm);
  }
}

public class WeatherReport {
  [Frozen] private TempMeasurement temp;
  [Frozen] private WindMeasurement wind;
  public Location loc;

  invariant temp == null || loc == temp.loc;
  invariant wind == null || loc == wind.loc;

  public WeatherReport(Location loc)
    ensures this.loc == loc;
  { this.loc = loc; }
```

```
  public void AddTemp([Frozen] TempMeasurement! tm)
    requires loc == tm.loc;
    ensures loc == temp.loc;
    modifies this.temp;
  { temp = tm; }

  public void AddWind([Captured] WindMeasurement! wm)
    requires loc == wm.loc;
    ensures loc == wind.loc;
    modifies this.wind, wm.*;
  {
    wm.CorrectMeasurement();
    wind = wm;
  }

  [return: Frozen]
  public TempMeasurement GetTemp() {
    return temp;
  }

  [return: Frozen]
  public WindMeasurement GetWind() {
    return wind;
  }
}

public class CurrentWeather {
  [Rep] private TempMeasurement! temp;
  [Rep] private WindMeasurement! wind;

  public CurrentWeather(Date d, Location l) {
    temp = new TempMeasurement(d, l, null, null);
    wind = new WindMeasurement(d, l, null);
  }

  public void UpdateTemp(Date d, Temp airT, Temp waterT) {
    expose (this) {
      temp.date = d;
      temp.airTemp = airT;
      temp.waterTemp = waterT;
    }
  }

  public void UpdateWind(Date d, Velocity windVel) {
    expose (this) {
      wind.date = d;
      wind.windVelocity = windVel;
    }
  }
}

public class Measurement {
  public Date date;
  public Location loc;
```

```
  public Measurement(Date d, Location l)
    ensures date == d && loc == l;
  {
    date = d;
    loc = l;
  }

  public virtual void CorrectMeasurement()
    ensures loc == old(loc);
  {
    // check validity of measurement and fill in missing information
  }
}

public class TempMeasurement : Measurement {
  public Temp airTemp;
  public Temp waterTemp;

  public TempMeasurement(Date d, Location l, Temp airT, Temp waterT)
    ensures date == d && loc == l && airTemp == airT && waterTemp == waterT;
  {
    airTemp = airT;
    waterTemp = waterT;
    base(d, l);
  }

  public override void CorrectMeasurement() {
    // check validity of measurement and fill in missing information
  }
}

public class WindMeasurement : Measurement {
  public Velocity windVelocity;

  public WindMeasurement(Date d, Location l, Velocity windVel)
    ensures date == d && loc == l && windVelocity == windVel;
  {
    windVelocity = windVel;
    base(d, l);
  }

  public override void CorrectMeasurement() {
    // check validity of measurement and fill in missing information
  }
}
```

A.2 Traveler

This section contains the *Traveler* example that has been used in a simplified version in the paper originally introducing frozen objects [7] and has been adapted here to conform to the full Spec# language.

```
[Immutable]
public class City {
  public string! name;

  public City(string! name) {
    this.name = name;
  }
}

[Immutable]
public class Route {
  public City! from;
  public City! to;

  public Route(City! from, City! to) {
    this.from = from;
    this.to = to;
  }
}

public class Map {
  [Rep] private List<City>! cities = new List<City>();
  [Rep] private List<Route>! routes = new List<Route>();

  public void AddCity(City! city)
    modifies cities.*;
  {
    expose (this) {
      if (!cities.Contains(city))
        cities.Add(city);
    }
  }

  public void AddRoute(Route! route)
    modifies routes.*;
  {
    expose (this) {
      if (!routes.Contains(route))
        routes.Add(route);
    }
  }

  [Pure]
  public City GetCity(string! name) {
    City city = null;
    foreach (City c in cities) {
      if (c != null && c.name.Equals(name))
        city = c;
    }
    return city;
  }
```

```
  [Pure]
  public bool HasRoute(City! from, City! to)
    ensures from == to ==> result == true;
  {
    foreach (Route r in routes) {
      if (r != null && r.from == from && r.to == to) return true;
    }

    if (from == to) return true;
    return false;
  }
}

public class PublicTransport {
  public void AddTrainLines(Map! map)
    modifies map.*;
  {
    City c0 = map.GetCity("Zürich");
    City c1 = map.GetCity("Milano");
    City c2 = map.GetCity("Roma");

    if (c0 != null && c1 != null) map.AddRoute(new Route(c0, c1));
    if (c1 != null && c2 != null) map.AddRoute(new Route(c1, c2));
  }

  public void AddBusLines(Map! map)
    modifies map.*;
  {
    City c0 = map.GetCity("Zürich");
    City c1 = map.GetCity("Kloten");

    if (c0 != null && c1 != null) map.AddRoute(new Route(c0, c1));
  }
}

public class Traveler {
  [Frozen] private Map! map;
  private City! current, next;

  invariant map.HasRoute(current, next);

  public Traveler([Frozen] Map! map, City! current) {
    this.map = map;
    this.current = current;
    this.next = current;
  }

  public void NextTarget(City! target) {
    if (map.HasRoute(current, target)) {
      expose(this) {
        next = target;
      }
    }
  }
```

```
  public void MakeStep() {
    expose(this) {
      current = next;
    }
  }
}

public class Main {
  public void Setup(PublicTransport! pt)
    modifies pt.*;
  {
    City c0 = new City("Zürich");
    City c1 = new City("Kloten");
    City c2 = new City("Milano");
    City c3 = new City("Roma");

    Map map = new Map();
    map.AddCity(c0);
    map.AddCity(c1);
    map.AddCity(c2);
    map.AddCity(c3);

    pt.AddTrainLines(map);
    pt.AddBusLines(map);

    freeze map;

    Traveler t0 = new Traveler(map, c0);
    Traveler t1 = new Traveler(map, c1);

    t0.NextTarget(c3);
    t1.NextTarget(c0);
    t0.MakeStep();
    t1.MakeStep();
  }
}
```

Classes *City* and *Route* are both declared as [Immutable]. Class *Map* contains a list of cities and a list of routes between those cities. We can add a city or a route to a map, retrieve a city from it, or ask whether a specific route belongs to a map. With the methods in class *PublicTransport*, we can add train or bus lines to a given map.

Class *Traveler* takes a frozen map and a city as parameters to its constructor and initializes the fields *map*, *current*, and *next* with them. In this class, we make use of being allowed to write an invariant over a frozen field. This invariant guarantees that there is always a route in the map between the cities stored in fields *current* and *next*. The constructor of *Traveler* satisfies this invariant by initializing both *current* and *next* to the same city. Method *NextTarget*, which sets field *next* to a new city, also retains the invariant by only modifying the field, when there is a route between *current* and the new *target* city. Method *MakeStep* sets the *current* field to the city stored in *next*, thereby also maintaining the class invariant.

In method *Setup* of class *Main*, we create four new cities and add them to a map. Next, we add train and bus lines to the map by calling the corresponding methods on the *PublicTransport* object that we get as parameter. Then we freeze the map and give it to two new travelers, which have a different starting city, may both independently set their next targets, and make steps while still maintaining their invariant over the map.

Appendix B

Changes in Preexisting Test Cases

While integrating the frozen objects methodology into Spec# and Boogie, we have taken great care that the implementation not only satisfies all our new test cases, but also that all the preexisting test cases still work correctly. However, in some cases this has not been possible, mainly due to the fact that objects of an immutable type have been unowned before and are now owned by the *Freezer*.

We have observed three types of changes in the outcome of the test cases with the old and the new implementation. In Section B.1, we discuss errors that get detected in the old as well as in the new implementation but with different error messages. In Section B.2, we look at error messages that only exist in the new implementation, and in Section B.3, we discuss errors in the old implementation that are no longer errors in the new implementation.

B.1 Different Error Messages for Same Error

Immutability of objects has been a concept completely separate from the ownership model in the previous implementation, and there has been a set of error messages specifically for verification errors of immutable objects. With the new implementation, we do not need these additional error checks anymore, because violations of frozen object properties are detected with the already existing checks of the ownership model.

The following example shows two methods that assign an object of an immutable type to a peer and a rep field of type *object* respectively:

```
public class Foo {
  [Rep] object or;
  [Peer] object op;

  public void Bar1(ImmutableClass ic) {
    or = ic;
  }

  public void Bar2(ImmutableClass ic) {
    op = ic;
  }
}

[Immutable]
public class ImmutableClass { }
```

Both assignments are illegal, as we are not allowed to assign an object of an immutable type to a field specified as [Peer] or [Rep]. In the old implementation, there have been special checks

to uncover these errors, and the error messages have been as follows:

```
Error: RHS might be immutable, and not allowed in rep fields
Error: RHS might be immutable, and not allowed in peer fields
```

In the new implementation, we do not need these special checks anymore, and the errors get detected by the checks of the ownership model:

```
Error: assigning the owner of this object may violate the modifies clause
Error: illegal assignment to rep field, RHS may already have a different owner
Error: assigning the owner of this object may violate the modifies clause
Error: illegal assignment to peer field, target object and RHS may have
    different owners or an owner might not be exposed
```

We get two error messages for each error, the first stating the assignment may violate the modifies clause, because we are not allowed to modify the owner of *ic* without declaring the parameter as [Captured], and the second error states that the object on the right-hand side (RHS) may already have a different owner, which in this case is the *Freezer* object.

B.2 New Errors

Objects of an immutable type have been defined as peer consistent in the old implementation. However, now that such objects are owned by the *Freezer*, they cannot be peer consistent anymore, because their owner can never be exposed. Test cases in which the peer consistency of objects of an immutable type has been verified before, therefore, return an error message now with the new implementation.

For example, a method that expects a parameter of type *object* requires this parameter to be peer consistent by default. In the old implementation, we could give a string as parameter to such a method, but a string is never peer consistent in the new implementation, as *string* is an immutable type and, therefore, all its instances are frozen. Test cases where a string (or any other object of an immutable type) is provided to a method expecting a parameter of type *object*, hence, return an additional error message now.

B.3 Previous Errors

There are also some cases, where we got an error message in the old implementation, but do not get an error message anymore now. The following example shows two of those cases:

```
public class Foo {
  [Peer] object op;

  public void Bar1([Captured] object obj) {
    op = obj;
  }

  public void Bar2([Captured] object! x, object! y) {
    Owner.AssignSame(x, y);
  }
}

[Immutable]
public class ImmutableClass { }
```

When we verify method *Bar1* of this example with the old implementation of Boogie, we get the following error message:

```
Error: RHS might be immutable, and not allowed in peer fields
```

As type *object* is the only type that can have mutable and immutable subtypes, there has been no way of statically knowing whether the provided parameter is immutable. Hence, an error has been thrown, because an object of an immutable type is not allowed to be assigned to a peer field.

With the new frozen objects methodology, method *Bar1* can be verified without an error. The [Captured] modifier requires parameter *obj* to be unowned, so we can be sure that *obj* is never frozen and, therefore, is never of an immutable type.

Method *Bar2*, verified with the old implementation of Boogie, results in the following two error messages:

```
Error: the given peer object might be immutable, and its owner is not allowed to
    be assigned
Error: the subject might be immutable, and its owner is not allowed to be
    assigned
```

Both x and y could potentially be of an immutable type, and an object of such a type is neither allowed to be assigned an owner, nor may the owner of such an object be assigned to another object.

With the new implementation, we again get no error messages, because x is specified as [Captured] and, therefore, guaranteed to be unowned, and the owner of y is allowed to be assigned as the owner of x even if it is the *Freezer*.

These examples show that in some cases the new frozen objects methodology is stronger than the old implementation. Before, we got error messages just in case the object is of an immutable type. Now, we only get an error, if the method is actually called with such an object as parameter, i.e. as long as we avoid calling those methods with an object of an immutable type, the verification succeeds.

Appendix C

Uncovered Bugs

While implementing the frozen objects methodology into Spec#, we uncovered and fixed some bugs in the code of Spec#, Boogie, and the *Prelude* file. In the following sections, we describe the issues and our solution.

C.1 Spec#

So far, declaring a field as [Peer] and [Rep] at the same time has been legal. When verifying a program with such a field, Boogie just ignored the second modifier. We changed the implementation, so that we now get a compiler error, when a field is declared as [Peer] and [Rep], same as we also get an error by declaring a field as [Frozen] together with another ownership modifier.

C.2 Prelude

When freezing an object, we always have to check for peer consistency first. We found that after a call to a pure function, the prover could suddenly assert peer consistency for a frozen object, whose owner clearly cannot be exposed as would be required for being peer consistent. This problem has been caused by an error in the *Prelude* file. The function *##FieldDependsOnFCO* has been implemented as follows:

```
function ##FieldDependsOnFCO<alpha>(o: ref, f: Field alpha, ev:
    exposeVersionType) returns (exposeVersionType);
```

The return type of this function has been set incorrectly and we therefore had to change it to:

```
function ##FieldDependsOnFCO<alpha>(o: ref, f: Field alpha, ev:
    exposeVersionType) returns (alpha);
```

C.3 Boogie

The method *IsImmutable* of class *Sink* has failed to recognize immutability in the presence of non-null types, arrays, and generic types. We first tried to circumvent this problem by adding special cases to the method, but then we realized that only one statement has been missing:

```
internal static bool IsImmutable (Cci.TypeNode! type, out bool isImmutable) {
  Cci.TypeNode tn = Cci.TypeNode.StripModifiers(type);

  ...
}
```

Non-null types, arrays, and generic types are nested. For example, a non-null type is a type with a non-null modifier and a reference to the actual type. Hence, modifiers that are applied to the actual type, like [Immutable], cannot be directly retrieved for non-null types by calling *GetAttribute*. However, if we call the method *StripModifiers* first, then all those types get unnested, and we can retrieve its modifiers as with any other type.

Bibliography

[1] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. *The Spec# programming system: An overview.* In LNCS, volume 3362. Springer, 2004.

[2] K. Rustan M. Leino, and Peter Müller. *Object Invariants in Dynamic Contexts.* In LNCS, volume 3086. Springer, 2004.

[3] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. *Boogie: A modular reusable verifier for object-oriented programs.* In LNCS, volume 4111. Springer, 2006.

[4] K. Rustan M. Leino. *This is Boogie 2.* Manuscript KRML 178. 2008.

[5] K. Rustan M. Leino, and Angela Wallenburg. *Class-Local Object Invariants.* In ISEC, pages 57-66. ACM, 2008.

[6] Leonardo de Moura, and Nikolaj Bjrner. *Z3: An Efficient SMT Solver.* In LNCS, volume 4963. Springer, 2008.

[7] K. Rustan M. Leino, Peter Müller, and Angela Wallenburg. *Flexible Immutability with Frozen Objects.* In VSTTE, volume 5295. Springer, 2008.

[8] K. Rustan M. Leino. *Spec# Object Primer.*
http://channel9.msdn.com/wiki/specsharp/specsharpobjectprimer.

[9] Microsoft Research. *Open Source Spec#.*
http://specsharp.codeplex.com.