Flexible Immutability with Frozen Objects

K. Rustan M. Leino[0], Peter Müller[0], and Angela Wallenburg[1]

[0] Microsoft Research, {leino,mueller}@microsoft.com
[1] Chalmers University of Technology and Göteborg University, angelaw@chalmers.se

Abstract. Object immutability is a familiar concept that allows safe sharing of objects. Existing language support for immutability is based on immutable classes. However, class-based approaches are restrictive because programmers can neither make instances of arbitrary classes immutable, nor can they control when an instance becomes immutable. These restrictions prevent many interesting applications where objects of mutable classes go through a number of modifications before they become immutable.

This paper presents a flexible technique to enforce the immutability of individual objects by transferring their ownership to a special freezer object, which prevents further modification. The paper demonstrates how immutability facilitates program verification by extending the Boogie methodology for object invariants to immutable objects. The technique is based on Spec#'s dynamic ownership, but the concepts also apply to other ownership systems that support transfer.

0 Introduction

Object immutability is a familiar concept that allows safe sharing of objects. For instance, immutable objects can be accessed concurrently without locking, and the absence of state changes can simplify reasoning about pointer structures.

A simple recipe for implementing immutable objects is the following: (0) Encapsulate the state of the immutable object by hiding all mutable fields from clients. (1) Do not provide any methods in the immutable object that modify the state of their receiver. (2) Encapsulate the mutable sub-objects of immutable aggregate objects by ensuring that references to mutable sub-objects do not escape from the aggregate.

This recipe is for instance applied in Java's *String* class: Its fields are private or final, methods that manipulate strings create new instances, but do not modify their receiver, and methods do not leak references to the (mutable) array that forms the internal representation of a string. The recipe is also amenable to static checking [12].

Even though this recipe is useful for value objects such as strings, it is too restrictive for many advanced applications of immutability. First, the recipe restricts *when an object becomes immutable*. Since immutable objects must not have mutating methods (rule 1), immutable objects must be fully initialized when their constructor terminates. This requirement prevents common initialization schemes such as multi-phase initialization. Second, the recipe restricts *the classes whose instances may be used as immutable objects*. Most classes in a program violate rule 1 of the recipe because they provide mutating methods. Nevertheless, it is often useful to have immutable instances of such classes, for instance, immutable instances of general collections or, similarly, immutable arrays. The standard workaround for this limitation of the recipe is to wrap the supposedly-immutable instance or array in an immutable object that actually does

follow the recipe. However, this workaround requires extra code to delegate calls to the wrapped object and also requires very careful design to ensure that there are no aliases to the wrapped object after it has been wrapped.

```
using Map := Graph⟨City⟩ ;

class City {
  // some state, e.g. population
  City(String name)
  { /* code omitted */ }
}

class Graph⟨N⟩ {
  void AddNode(N n)
    modifies this.* ;
  { /* code omitted */ }
  void AddEdge(N from, N to)
    modifies this.* ;
  { /* code omitted */ }
  bool HasEdge(N from, N to)
  { /* code omitted */ }
  List⟨N⟩? Path(N from, N to)
  { /* code omitted */ }
}

class Metro {
  void AddTrainLines(Map map) ;
    modifies map.* ;
  { /* code omitted */ }
  void AddBusLines(Map map) ;
    modifies map.* ;
  { /* code omitted */ }
}

class Traveler {
  frozen Map map ;
  City current, next ;
  invariant map.HasEdge(current, next) ;
  Traveler(frozen Map map, City current)
  {
    this.map := map ;
    this.current := current ;
    this.next := current ;
  }
  void GoForward(City target)
    modifies this.* ;
  {
    List⟨City⟩? route ;
    route := map.Path(current, target) ;
    expose (this) {
      if (route = null) next := current ;
      else next := route.ItemAt(1) ;
    }
  }
}

class Main {
  void Setup(Metro metro)
  {
    Map map := new Map() ;
    metro.AddTrainLines(map) ;
    metro.AddBusLines(map) ;
    freeze map ;
    City c0 := new City("Oslo") ;
    City c1 := new City("Rome") ;
    Traveler t0 := new Traveler(map, c0) ;
    Traveler t1 := new Traveler(map, c1) ;
    /* ... */
  }
}
```

Fig. 0. Running example in a Spec#-like language. We assume that reference types are non-null types unless indicated otherwise by appending a question mark. The **using** directive introduces a synonym for a type. Frame specifications using **modifies** clauses indicate the potential side effects of a method. The **frozen** modifier as well as the **freeze** and **expose** statements are part of our methodology and will be explained later in the text.

Example. We illustrate the limitations of the above recipe as well as our approach using the running example in Fig. 0. The example models travelers (class $Traveler$) who navigate using a map, which is a graph whose nodes are cities (abbreviated by type name Map). The map is shared by all travelers. We use an immutable map to allow $Traveler$ to maintain a non-trivial invariant, namely that the map contains an edge between the traveler's current city and the next city on their route.

This example does not follow the above recipe. First, the map undergoes a complex initialization phase. After it is created in method $Setup$ (class $Main$), it is passed as argument to two methods of class $Metro$, where more edges are added to the map. Only after these additions does the map become immutable. Such complex initialization phases occur frequently. For instance, nodes of an abstract syntax tree are mutated during resolution and type checking, but are often immutable afterwards. Second, the map is an instance of class $Graph$, which provides mutating methods such as $AddEdge$. Nevertheless, sharing of a map among travelers should be permitted, because travelers do not call these mutating methods.

Approach and Contributions. In this paper, we present a programming methodology that gives programmers the flexibility to decide when an object becomes immutable and also supports immutable instances of mutable classes.

Our work builds on the Boogie methodology for the verification of object invariants [1, 13]. This methodology arranges objects in an ownership hierarchy and controls modifications of objects. It imposes the rule that an object must be *exposed* before its fields can be modified, and the rule that an object can be exposed only after its owner has been exposed. These two rules guarantee that modifications of objects are always initiated from the root of an ownership tree. Using this methodology, we can support immutable objects using three key ideas:

0. We use ownership to delimit the portion of an object's state that is immutable. In other words, the immutable state of an object o comprises the fields of o as well as the state of all objects (transitively) owned by o.
1. We provide a *freeze* operation that turns an object into an immutable object. This operation is an ownership transfer to a designated owner, called *freezer*, which cannot be referred to by the program. The freeze operation lets a programmer decide when an object becomes immutable.
2. We ensure that the freezer and all objects it owns, cannot be exposed. Consequently, the Boogie methodology prevents all modifications of the frozen objects, which makes them immutable even if they contain mutating methods or public fields.

In this paper, we present the details of this approach using Spec#'s dynamic ownership [13]. Our methodology also works with static ownership type systems that support ownership transfer [0,6, 20].

Our work is related to various type systems for immutable objects [4, 12, 22, 24], which also use ownership to delimit the object state and to control modifications. However, our methodology builds on verification instead of a type system to provide extra flexibility that we needed in several verification case studies. We also extend the Boogie methodology to make use of immutable objects. In summary, the two main contributions of our work are:

0. A verification methodology that supports immutability on a per-object basis. We show that immutable types are a special case of our methodology, where each instance is frozen at a particular program point. Our methodology also supports immutable instances of mutable types such as arrays.
1. An extension to the Boogie methodology that (a) guarantees that immutable objects satisfy their invariants in all states and (b) allows invariants to depend on immutable shared objects. We present the axioms and specification idioms for this extension.

Outline. The next two sections (1 and 2) explain the details of frozen objects and show how they subsume immutable types. Sec. 3 presents applications of frozen objects for the verification of object invariants. Sec. 4 discusses the application of frozen objects in Spec#. We discuss related work in Sec. 5 and conclude in Sec. 6.

1 Frozen Objects

In this section, we provide the background on the Boogie methodology that is needed in the rest of the paper. Based on this methodology, we explain how objects can be frozen and how frozen objects are encoded in the program logic. Since it is orthogonal to immutability, we defer a discussion of subtyping until Sec. 4.

1.0 Background on Boogie Methodology

The Boogie methodology [13] enables the sound verification of object invariants in the presence of callbacks by tracking whether an object is *valid*, that is, its invariant is known to hold, or whether it is *mutable*, that is, its invariant is allowed to be broken. Whether an object is valid or not is stored in a boolean field inv. The Boogie methodology maintains the following program invariant in all execution states:

PI0: $(\forall\, o \bullet\ o.inv \Rightarrow Inv(o)\ )$

Here and throughout, the quantifier reaches over all allocated, non-null objects, and $Inv(o)$ denotes the object invariant of object o.

New objects start off as mutable because their invariants have not yet been established. The program invariant is maintained by two fundamental rules: First, we enforce through a proof obligation that only fields of mutable objects can be assigned to; therefore, the assignments trivially maintain PI0. Second, the invariant $Inv(o)$ is checked before an object o becomes valid, that is, when $o.inv$ is changed to true.

Objects become valid at the end of their constructor. Moreover, programs can change the inv field through a designated **expose** block statement. $\mathbf{expose}\ (o)\{S\}$ sets $o.inv$ to false, thereby enabling modifications of o's fields. Then it executes statement S, checks o's invariant, and finally sets $o.inv$ back to true.

Aggregate objects are handled using ownership [7]. The invariant of an aggregate object o may depend on fields of another object x if x is owned by o[0]. The Boogie methodology enforces that the objects owned by a valid object o are also valid:

[0] The Boogie methodology supports several other kinds of multi-object invariants. We discuss invariants over peers [13] in Sec. 4. An extension of our methodology to history invariants [15] is straightforward.

PI1: $(\forall\, o, x \bullet\ x.owner = o \wedge o.inv \Rightarrow x.inv\ )$

This program invariant is enforced by checking that the owner o of an object x is mutable when x is being exposed. It makes sure that program invariant $PI0$ is preserved even though modifications of x potentially break the invariant of o.

The Boogie methodology uses a dynamic encoding of ownership as opposed to a static type system. Each object has a field $owner$ that holds a reference to the (single) owner object, or **null** if the object has no owner. The ownership relation is expressed by putting **rep** annotations on field declarations. For a field f, such an annotation gives rise to the implicit invariant $\mathbf{this}.f \neq \mathbf{null} \Rightarrow \mathbf{this}.f.owner = \mathbf{this}$. However, the $owner$ field must not be used in explicit object invariants.

The $owner$ field of an object x can be set to an object o by the special statement $\mathbf{transfer}\ x\ \mathbf{to}\ o$. Since the transfer might break the implicit ownership invariant of the previous owner, the **transfer** statement requires x's previous owner, if any, to be mutable. This requirement ensures that $PI0$ is maintained. To maintain $PI1$, **transfer** requires that x be valid or that the new owner o be **null** or mutable.

```
x.f := e  ≡
  assert x ≠ null ∧ ¬x.inv ;
  x.f := e ;

expose (o) { S }  ≡
  assert o ≠ null ∧ o.inv ;
  assert o.owner ≠ null ⇒ ¬o.owner.inv ;
  o.inv := false ;
  S ;
  assert (∀ x • x.owner = o ⇒ x.inv ) ;
  assert Inv(o) ;
  o.inv := true ;

transfer x to o  ≡
  assert x ≠ null ;
  assert x.owner ≠ null ⇒
         ¬x.owner.inv ;
  assert x.inv ∨ o = null ∨ ¬o.inv ;
  x.owner := o ;

freeze x  ≡
  assert x ≠ null ;
  assert x.owner ≠ null ⇒
         ¬x.owner.inv ;
  assert x.inv ;
  x.owner := freezer ;
```

Fig. 1. Pseudo code for field update, **expose**, **transfer**, and **freeze**. The new **freeze** statement is a variation of ownership transfer and will be explained in Sec. 1.1.

We define the semantics of the **expose** and **transfer** statements by translating them into the pseudo code shown in Fig. 1. Proving the correctness of a program amounts to statically verifying that the program does not abort due to a violated assertion. One can use an appropriate program logic to show that the assertions hold.

A typical method reads the state of its receiver and parameters—either by inspecting the state or by calling methods that do—and expects the invariants of these objects to hold. Therefore, we define a default precondition for all methods that requires their receiver and parameters to be valid. A common way to satisfy this default precondition is to use ownership—objects owned by a valid object are also known to be valid ($PI1$).

If a method modifies the state of an object, it needs to declare that modification in the method's **modifies** clause. We then say that the method *mutates* the object and that the method is *mutating*. We say that a method with an empty **modifies** clause is

pure. To perform the actual mutation of an object, a method needs to first expose the object, which requires the object's owner to be mutable. Therefore, for every object that a mutating method mutates (that is, that it lists in its **modifies** clause), we define an additional default precondition that requires that object's owner to be mutable.

In method $Main.Setup$, the call to the mutating method $AddTrainLines$ satisfies its default precondition because $Setup$'s default precondition guarantees $metro$ to be valid, and the **new** allocation constructs map to be valid and without an owner.

1.1 The Freeze Statement

In the Boogie methodology, an object can be modified only if the object and all its transitive owners are mutable. This offers a simple way of enforcing immutability. We introduce an object $freezer$, which is valid in all program states. The $freezer$ object cannot be referred to in the program. In particular, it cannot be exposed. Consequently, objects owned by the $freezer$, called *frozen objects*, also cannot be exposed (by the second assertion of **expose**, see Fig. 1) and, thus, cannot be modified (by the assertion for field updates, see Fig. 1).

We provide a statement **freeze** x, which takes a valid object x and sets $x.owner$ to the $freezer$. Since the $freezer$ is always valid, the pseudo code for **freeze** is a bit simpler than for **transfer**, see Fig. 1. Freezing an object x affects only one field, namely $x.owner$. The operation maintains program invariant $PI0$, because the only object invariants that can depend on $x.owner$ are the (implicit) object invariants of x's owner, and the precondition checks that any such owner is mutable. It also maintains $PI1$, because $x.inv$ is asserted before x is assigned a new owner.

The **freeze** statement allows programmers to choose when an object becomes immutable. For simple value objects such as strings, this will be at the end of the constructor. But the **freeze** statement may also occur later, for instance, after a complex initialization phase as illustrated by method $Main.Setup$. The **freeze** map operation succeeds because map is valid and has no owner. Any subsequent attempt to expose map would fail because the second precondition of **expose** map does not hold: map does have an owner, namely the freezer, and the owner is valid. For the same reason, the object cannot be un-frozen by transferring it to another owner because the second assertion in the pseudo code for **transfer** would fail. In other words, the immutability of an object also applies to its $owner$ field, and ownership transfer is just an update of $owner$. Consequently, map is permanently immutable once it has been frozen.

It is illustrative to discuss how frozen objects replace the recipe for immutable objects presented in the introduction. Rule 0 becomes dispensable because a frozen object cannot be exposed and, thus, its fields cannot be updated. For the same reason, the default precondition of mutating methods cannot be established, which makes rule 1 dispensable. This allows programs to create immutable instances of any class, even those that provide mutating methods such as $Graph$. Finally, rule 2 becomes dispensable because the immutability of a frozen object also applies transitively to the objects it owns. Assume for instance that a $Graph$ is represented by a set of $Node$ objects, which are owned by the $Graph$ object. The nodes of a frozen $Graph$ object are transitively owned by the freezer. Consequently, they cannot be exposed, which makes them immutable, even if they are leaked outside the graph structure.

1.2 Writing Specifications about Frozen Objects

To exploit properties of frozen objects during verification, specifications need to express which objects are frozen. For this purpose, we introduce a boolean function $\mathit{frozen}(o, h)$ that yields whether an object o is (transitively) owned by the freezer in heap h. We omit the heap parameter whenever it is clear from the context.

Instead of using frozen directly in specifications, we allow fields, method parameters, and method results to be declared with the modifier $\mathbf{frozen}$. For a field f, this modifier gives rise to the implicit invariant $\mathbf{this}.f \neq \mathbf{null} \Rightarrow \mathit{frozen}(\mathbf{this}.f)$. This invariant, like all other object invariants, is checked at the end of constructors and $\mathbf{expose}$ blocks. For method parameters and results, we introduce analogous pre- and postconditions. For instance, class $\mathit{Traveler}$ uses the $\mathbf{frozen}$ modifier in the declaration of field map and the constructor's first parameter.

For the soundness of the Boogie methodology, it is essential that each object invariant is admissible, that is, that it does not compromise program invariant $PI0$. Since the owner field of a frozen object x is immutable, $\mathit{frozen}(x)$ can never change from true to false. Thus, the implicit invariant for a frozen field f in an object o can be broken only by updating $o.f$. This update requires o to be mutable and, thus, preserves $PI0$.

1.3 Formal Encoding

The encoding of frozen objects in a program logic formalizes which objects are frozen and that frozen objects are immutable.

Frozen Objects. To capture the first aspect, one could define $\mathit{frozen}(o, h)$ to hold if and only if o is transitively owned by the freezer in heap h. However, automatic theorem provers such as Simplify and Z3 perform poorly on transitive closure. Therefore, we provide a weaker axiomatization of frozen, which is sufficient for practical examples: (a) The freezer itself is frozen. (b) Objects directly owned by a frozen object are also frozen:

$$(\forall\, h \bullet\ \mathit{frozen}(\mathit{freezer}, h)\ ) \tag{0}$$

$$(\forall\, o, h \bullet\ o.\mathit{owner} \neq \mathbf{null} \wedge \mathit{frozen}(h[o, \mathit{owner}], h) \Rightarrow \mathit{frozen}(o, h)\ ) \tag{1}$$

$h[o, \mathit{owner}]$ denotes the value held by field owner of object o in heap h. We abbreviate this notation by $o.\mathit{owner}$ when the heap is clear from the context.

Axioms (0) and (1), in combination with the pseudo code for the $\mathbf{freeze}$ statement and the $\mathbf{frozen}$ modifier explained in the previous subsection, allow one to prove that certain objects are frozen in a given heap. For instance, one can prove that $\mathit{frozen}(\mathit{map})$ holds in the heap after the execution of $\mathbf{freeze}$ map in method Setup.

We have argued above that an object that is frozen in a heap h remains frozen in all successor heaps h' of h. We encode this property using a relation $\mathit{HeapSucc}(h', h)$ that expresses that heap h' is a successor of heap h. For every object allocation and field update, we add an assumption that the resulting heap is a successor of the initial heap. Moreover, we add a postcondition to each method and constructor stating that the poststate heap is a successor of the prestate heap. This postcondition may be assumed

by callers, but need not be proven for method or constructor implementations. Using $HeapSucc$, we can now state that frozen objects remain frozen by the following axiom:

$$(\forall\, o, h, h' \bullet\ HeapSucc(h', h) \wedge frozen(o, h) \Rightarrow frozen(o, h')\ ) \tag{2}$$

In our example, this axiom allows us to prove that the allocation and initialization of $City$ and $Traveler$ objects in method $Setup$ do not invalidate $frozen(map)$.

Immutability of Frozen Objects. We encode immutability of frozen objects by an axiom that says that the value of a field of a frozen object is a function of the object reference and the field name alone, and in particular does not depend on the heap:

$$(\forall\, o, f, h \bullet\ frozen(o, h) \Rightarrow h[o, f] = \#UltimateValue(o, f)\ ) \tag{3}$$

where $\#UltimateValue$ is an uninterpreted function symbol.

To illustrate the use of this axiom, we prove that the value of some field $map.nodes$ is the same before and after allocating and initializing the $City$ objects in method $Setup$. Let h and h' denote the heaps in these states. We show this property by instantiating axiom (3) twice, once with map, $nodes$, and h, and once with map, $nodes$, and h'. As argued above, we can show $frozen(map, h)$ as well as $frozen(map, h')$. So we derive:

$$h[map, nodes] = \#UltimateValue(map, nodes) \quad \text{and}$$
$$h'[map, nodes] = \#UltimateValue(map, nodes)$$

which trivially implies the desired $h[map, nodes] = h'[map, nodes]$.

2 Immutable Types

With frozen objects, we can designate individual objects as being immutable from a particular time onward. Therefore, frozen objects are strictly more general than immutable types, where the immutability of an object is determined by its type and always occurs from the end of the constructor on. In this section, we show how frozen objects can be used to encode immutable types and explain the benefits of such an encoding.

2.0 Encoding Immutable Types

Let's assume that immutable classes are marked with a modifier $\mathbf{immutable}$. We define the meaning of an $\mathbf{immutable}$ class C in terms of what we introduced in the previous section. An allocation $c := \mathbf{new}\ C(\ldots)$ is immediately followed by an implicit statement $\mathbf{freeze}\ c$. Therefore, every instance of class C is immutable once its constructor has terminated.

To make use of the immutability, for instance, for verification, we augment specifications as follows: Every parameter, receiver, and return value p of static type C, except the receiver of constructors (and other *delayed* arguments, if applicable [11]), gives rise to an implicit precondition (or postcondition, in the case of results) $p \neq \mathbf{null} \Rightarrow frozen(p)$, as if p had been declared with the $\mathbf{frozen}$ modifier. Similarly, every field

f of static type C gives rise to the implicit object invariant $f \neq \mathbf{null} \Rightarrow \mathit{frozen}(f)$, as if the field had been declared with the **frozen** modifier. Finally, every local variable x of static type C gives rise to an implicit loop invariant $x \neq \mathbf{null} \Rightarrow \mathit{frozen}(x)$ on every loop that modifies x. These implicit pre- and postconditions and invariants are checked in the same way that explicit ones are. These checks, for instance, prevent a program from passing an instance of C as a (**frozen**) parameter before the instance has been fully initialized and frozen.

We also allow the **immutable** modifier to be applied to interface types, with the same meaning as just described for **immutable** classes. A class or interface that extends or implements an **immutable** class or interface must itself be declared **immutable**; therefore the object stored in a variable of an **immutable** type is actually an instance of an **immutable** class.

2.1 Benefits

The current implementation of immutable types in Spec# is not yet based on frozen objects and, although our motivation had been different from theirs, shares most virtues of the immutable-type design by Haack *et al.* [12]. In this subsection, we explain that our new design is not only more flexible, but has two additional major virtues compared to alternative designs.

A first virtue is that, rather than introducing the pre- and postconditions and invariants, as described in the previous subsection, it is tempting simply to encode an immutable type C by the following *Tantalizing Axiom*:

$$(\forall\, o, h \bullet\ o \in C \Rightarrow \mathit{frozen}(o, h)\ )$$

However, one has to be careful about the reach of o in this quantification, because an object of type C is allowed to undergo mutations (and in particular initialization) until the end of its construction. Various ways exist to exclude from the reach of the Tantalizing Axiom all objects currently being constructed. For example, one can design the programming language in such a way that the object being constructed does not come into being until values for all its fields have been computed (see, *e.g.*, Theta [17]). In languages like Spec# and Java, a constructor can assign to a field multiple times and can access the object while it is being constructed. To use the Tantalizing Axiom for such languages, one needs a stronger antecedent as well as some escape analysis in constructors. Defining this antecedent turned out to be complicated; we found that it was difficult to prevent the Tantalizing Axiom from kicking in too soon and interacting with other axioms in undesired ways. In contrast, the encoding of **immutable** types on top of frozen objects systematically introduces checks that objects of immutable types have reached their frozen state.

Another virtue of our encoding pertains to the knowledge that certain objects are not immutable. In a system where immutability is found only at the granularity of types, a checker must decide based on the static type of a target object o whether or not to allow an assignment to a field $o.f$, which in general depends on the dynamic type of o. For example, suppose the static type B of o were a mutable class and the program contained an immutable subclass C of B; then, a type-based analysis cannot determine precisely whether $o.f$ is allowed to be modified because at runtime, o might

reference an (immutable) C object. A sensible solution is to impose a *One-Down Rule* that says that each immediate subclass T of $Object$—the root of the class hierarchy—must either decide to make T and all its subclasses mutable or to make them all immutable [12]. In our new encoding of immutable types, checking if an assignment to $o.f$ is legal depends on the dynamic state of object o, and in particular on the validity of o. Therefore, we do not need such a One-Down Rule and instead allow mutable classes to have immutable subclasses (but not vice versa).

3 Frozen Objects and Object Invariants

Frozen objects have many benefits for writing and reasoning about code. For instance, they simplify the development of correct multi-threaded code and the verification of contracts containing pure method calls. In this section, we illustrate how frozen objects extend the benefits of owned objects—guaranteed validity and support for multi-object invariants—to shared objects.

3.0 Proving Validity

As explained in Sec. 1.0, the Boogie methodology makes explicit whether an object invariant may be assumed to hold. While making this information explicit enables the sound verification of object invariants in the presence of callbacks, it also complicates verification. Methods have default preconditions that require the validity of their arguments, and callers have to live up to these preconditions.

For instance, the call to $map.Path$ in method $Traveler.GoForward$ leads to a proof obligation that map is valid. In the Boogie methodology, validity of an object typically follows from ownership. If the $Traveler$ instance $\mathbf{this}$ owned $\mathbf{this}.map$ (that is, if map were a $\mathbf{rep}$ field), then the validity of $\mathbf{this}.map$ would follow from the validity of $\mathbf{this}$ by program invariant $PI1$. However, arranging for $Traveler$ objects to own their maps prevents them from sharing one map, which requires cloning of Map objects. For immutable objects, this cloning is unnecessary and unnatural because sharing is actually safe.

We solve this problem by encoding in our program logic that a frozen object is always valid. That is, we add the following axiom:

$$(\forall\, o, h \bullet\ frozen(o, h)\ \Rightarrow\ h[o, inv]\ ) \tag{4}$$

This axiom is justified because the $\mathbf{freeze}\ o$ statement requires o to be valid, and o cannot be exposed afterwards. Thus, o forever remains valid.

Axiom (4) allows one to show the validity of a frozen object without restricting sharing. For instance, from the implicit invariant for the $\mathbf{frozen}$ field map, we conclude that $\mathbf{this}.map$ is frozen in the prestate of method $GoForward$, and by axiom (4), we get $map.inv$. That is, we can share the Map instance among $Traveler$ objects and nevertheless live up to the default preconditions of methods operating on the map.

3.1 Invariants over Frozen Objects

Object invariants in the Boogie methodology are constrained by *admissibility requirements*. For the basic methodology (ownership-based invariants [1, 13]), an admissible invariant of an object o may depend on the state of o and of the objects (transitively) owned by o. This requirement ensures that all objects whose invariant is potentially broken by an update of $x.f$ are mutable such that program invariant $PI0$ is maintained.

While these ownership-based invariants enable modular verification of aggregate objects, they do not support sharing of objects. In our example, the invariant of class $Traveler$ depends on the state of a $Traveler$'s Map instance and would, therefore, be admissible only if map was a **rep** field. As we have explained in the previous subsection, this would prevent $Traveler$ objects from sharing a Map instance.

There are extensions to the basic Boogie methodology that support invariants over shared objects. However, these extensions restrict the classes of shared objects (visibility-based invariants [13]), restrict the invariants that one can write about shared objects [15], or complicate verification [2].

All variations of the Boogie methodology have in common that they restrict programs or invariants such that one can determine modularly all objects whose object invariant is potentially broken by an update of $x.f$; this allows one to impose proof obligations on the update that maintain program invariant $PI0$. Since frozen objects are immutable, their fields cannot be updated and, thus, maintaining object invariants over frozen objects is trivial.

To support such invariants, we relax the definition of admissible invariants of the basic Boogie methodology. In addition to the state of o and of the objects (transitively) owned by o, the invariant of an object o may now also depend on the state of frozen objects. This admissibility requirement can be checked syntactically by enforcing that only fields with the **rep** or **frozen** modifier can be dereferenced in an object invariant; such a syntactic check requires pure methods that are used in invariants to have read effect specifications [14].

In our example, the invariant of $Traveler$ is admissible under the assumption that the method $HasEdge$ reads only the state of its receiver and the objects owned by the receiver. Since field map is declared with the **frozen** modifier, we can conclude that the invariant depends only on the state of **this** and of frozen objects, namely the map referenced from $\mathbf{this}.map$.

The discussion above illustrates why it is not easily possible to permit frozen objects to be un-frozen. Un-freezing an object x would require one to find all objects whose invariants depend on the state of x because these invariants might be broken by modifications of x. Finding these objects in a modular way is at best complicated [2].

We have not yet implemented frozen objects in Spec#, but we manually changed the Spec#'s encoding of the program in Fig. 0 to resemble the encoding in this paper. The Boogie tool verified the resulting program successfully.

4 Application to Spec#

The form of the Boogie methodology implemented in Spec# is more advanced than the basic form we have presented here. In this section, we discuss how we have extended

our methodology to handle the three differences most relevant to this paper: updates without expose, subclasses, and peers.

Updates without Expose. One difference between the methodology we have presented here and what is implemented in Spec# is that Spec# is more lenient about when expose statements are needed. Instead of insisting that an object be mutable when one of its fields is updated, Spec# still allows the update if it maintains the object invariant. However, since object invariants of the owner may depend on the field being updated, and such object invariants might not be known in the current verification scope, Spec# will nevertheless check that the owner of the target object is mutable. Formally, for a field update of $o.f$, instead of the precondition $\neg o.inv$, Spec# actually checks:

$$\neg o.inv \vee ((o.owner \neq \mathbf{null} \Rightarrow \neg o.owner.inv) \wedge Inv'(o))$$

where $Inv'(o)$ denotes the invariant of o in the heap after the update of $o.f$.

Since the freezer is always valid, this more lenient field-update precondition always fails for a frozen object o. Consequently, fields of frozen objects do not change.

Subclasses. A second difference is that we have ignored subtyping so far whereas Spec# handles it. Instead of using just one inv field per object and letting the expose statement take an object argument, Spec# essentially uses one inv field per object-type pair and refines the expose statement to operate on object-type pairs as well [16]. Moreover, an owner in Spec# is not just an object but an object-type pair [13]. The effect of this support on the present paper is that the freezer becomes a pair $(freezer, Freezer)$, where $Freezer$ denotes some fixed but arbitrary type. Also, the condition $x.inv$ appearing in the preconditions of $\mathbf{transfer}$ and $\mathbf{freeze}$ (Fig. 1) is replaced by:

$$(\forall\ T \bullet\ (x, T).inv\ )$$

Peers. The third difference is that Spec# many times considers not just single objects but groups of peer objects. Two objects are *peers* if they have the same owner [13]. More precisely, in Spec#, the $owner$ field partitions objects into *peer groups*; the value of the $owner$ field is either the object-type owner (ow, T) that owns the objects in the peer group or, if the objects in the peer group have no owner, takes on the value $(r, \bot)$ where r is a representative element (object) of the peer group. Having an $owner$ value of the second kind corresponds to the condition that we have written $owner = \mathbf{null}$ elsewhere in this paper. Peer groups are similar to *clusters* [20] and allow the notion of peers even when objects are unowned.

In the presence of peer groups, a newly allocated object o starts off being unowned and belonging to a new peer group: $o.owner = (o, \bot)$. We redefine the statements $\mathbf{transfer}\ x\ \mathbf{to}\ o$ and $\mathbf{freeze}\ x$ to transfer not just x but the entire peer group of x; so the assignment to $x.owner$ in the pseudo code for these statements (Fig. 1) is replaced by the same assignment for all peers of x. If the new owner in the $\mathbf{transfer}$ statement is specified as $\mathbf{null}$, the right-hand side of the assignment is $(x, \bot)$. Also, the condition $x.inv$ appearing in the preconditions of these statements is replaced by:

$$(\forall\, p \bullet\ p.owner = x.owner \Rightarrow (\forall\, T \bullet\ (p, T).inv\ ))$$

which says that x is *peer valid*, meaning that x and all its peers are valid. (Note, though, that the precondition $o.inv$ of the **expose** statement remains what it is in Fig. 1, because the **expose** statement operates on a single object-type pair.)

In a similar way, Spec# does not use validity as the default method pre- and post-conditions we described it in Sec. 1.0, but uses peer validity. This makes no difference for frozen objects, since frozen objects are always peer valid.

5 Related Work

Immutability is a fundamental concept with many applications. In this section, we discuss work related to the two main contributions of this paper, how to enforce object immutability in imperative object-oriented languages and how to use immutable objects for verification.

Like our methodology, several type systems for immutability use ownership to delimit the state of an object. Boyapati [4] as well as Östlund *et al.* [22] present type systems that support immutable instances of mutable types and that let programmers decide when an object becomes immutable. Objects that will become immutable can only be referenced by unique references to make the transition to immutability type-safe. Since our methodology builds on verification rather than a type system, we do not need this restriction.

IGJ [24] uses Java's generic types to check immutability of classes, objects, and references. IGJ requires immutable objects to be fully initialized at the end of the constructor, which prevents complex initialization schemes. Oval [18] achieves per-object immutability by setting the owner to bottom. This is done when the object is created and, thus, not as flexible as frozen objects.

Haack *et al.* [12] present a type system for immutable types. As we have explained in detail in Sec. 2, our methodology is more flexible and, in fact, subsumes immutable types. A major virtue of Haack *et al.*'s type system is that it guarantees immutability even if some portions of the code (such as libraries) are not checked with the system. In contrast, we require all code to follow our methodology.

We presented our methodology in terms of Spec#'s dynamic ownership [13], but it can also be used with ownership type systems that support transfer such as Universes [20], External Uniqueness [6], or AliasJava [0] (even though AliasJava requires additional restrictions to prevent the modification of frozen objects through lent references).

Reference immutability like in Javari [23] or Universes [19] guarantees that certain read-only references are not used to modify an object. They allow safe sharing of objects. However, an object referenced through a read-only reference is still mutable and may be modified through other references. Thus, reference immutability does not have the same benefits for verification as immutable objects: objects referenced read-only cannot be assumed to be always valid, and object invariants must not depend on these objects [19]. In some of the examples we have encountered, we found use for an idiom similar to read-only references. We declared method parameters and fields as "peer or frozen". Like read-only references, such references cannot be used to modify the referenced object (because it might be frozen). However, unlike read-only references in

Universes, the object can still be assumed to be valid (the validity of peers follows from a method's peer-validity default precondition).

Boyland has used fractional permissions to differentiate read capabilities from write capabilities [5]. By squandering a part of an object's permissions, by giving a fractional permission to the freezer, or by weakening a full permission to an existentially quantified permission (cf. [3]), one can effectively freeze an object.

Visible state invariants may be assumed to hold in the pre- and poststates of all method executions; callers do not have to show the validity of method parameters explicitly. Thus, verification techniques for visible state invariants [10, 19] do not benefit from the fact that frozen objects are always valid. However, like the Boogie methodology, these techniques can be extended to support invariants over shared, frozen objects.

Our notion of immutability forbids all mutations of a frozen object, even benevolent mutations that are not observable by clients, such as lazy initialization. Naumann's work on observational purity [21] shows how to check that mutations are benevolent; it can be combined with our methodology to make it applicable to more programs.

6 Conclusions

The concept of immutable objects is useful and widely used in programming. Advanced support of immutability in a programming language or system requires a fine level of granularity, in order to accommodate the variety of ways that a program selects which objects are to be immutable and just when each object becomes immutable. In this paper, we have presented *frozen objects* as a technique for specifying and verifying programs that use such immutability properties. Our technique guarantees that the fields of frozen objects do not change. It is based on object ownership, where a special freezer object is used as the (transitive) owner of all frozen objects. Though we have presented our solution in the context of dynamic ownership, frozen objects also work with any ownership type system with ownership transfer.

Frozen objects subsume immutable types; encoding immutable types on top of frozen objects leads to a better axiomatization in a verification logic than a direct encoding and is also less restrictive.

Our technique guarantees that frozen objects are *valid*, meaning that their object invariants hold. In fact, this ever-validity had been our original motivation for this work. Because their fields do not change, frozen objects can be shared in a carefree way, and we allow any object invariant to depend on the fields of frozen objects.

There are other intriguing applications of immutability. One is in support of pure methods (*e.g.*, [9, 8, 14]), which for frozen objects return values that are insensitive to the heap. Another is in conjunction with concurrency (*e.g.*, [12]), which can benefit from the carefree sharing that frozen objects offer.

In future work, we intend to implement frozen objects in Spec#. We would also like to explore how to relax immutability to permit certain modifications of otherwise frozen objects, for instance, by using monotonicity.

References

0. J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *OOPSLA*, pages 311–330. ACM Press, 2002.
1. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *JOT*, 3(6):27–56, 2004. www.jot.fm.
2. M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, volume 3125 of *LNCS*, pages 54–84. Springer, 2004.
3. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL*, volume 40(1), pages 259–270. ACM, 2005.
4. C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. Ph.D., MIT, 2004.
5. J. Boyland. Checking interference with fractional permissions. In *SAS*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.
6. D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *ECOOP*, volume 2743 of *LNCS*, pages 176–200. Springer, 2003.
7. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, volume 33(10) of *ACM SIGPLAN Notices*, 1998.
8. Á. Darvas and K. R. M. Leino. Practical reasoning about invocations and implementations of pure methods. In *FASE*, volume 4422 of *LNCS*, pages 336–351. Springer, 2007.
9. Á. Darvas and P. Müller. Reasoning about method calls in interface specifications. *JOT*, 5(5):59–85, June 2006.
10. S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. A unified framework for verification techniques for object invariants. In *ECOOP*, LNCS. Springer, 2008.
11. M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *OOPSLA*, volume 42, number 10 in *SIGPLAN Notices*, pages 337–350. ACM, 2007.
12. C. Haack, E. Poll, J. Schäfer, and A. Schubert. Immutable objects for a Java-like language. In *ESOP*, volume 4421 of *LNCS*, pages 347–362. Springer, 2007.
13. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, volume 3086 of *LNCS*, pages 491–516. Springer, 2004.
14. K. R. M. Leino and P. Müller. Verification of equivalent-results methods. In *ESOP*, volume 4960 of *LNCS*, pages 307–321. Springer, 2008.
15. K. R. M. Leino and W. Schulte. Using history invariants to verify observers. In *ESOP*, volume 4421 of *LNCS*, pages 80–94. Springer, 2007.
16. K. R. M. Leino and A. Wallenburg. Class-local object invariants. In *First India Software Engineering Conference (ISEC)*. ACM, 2008.
17. B. Liskov, D. Curtis, M. Day, S. Ghemawat, R. Gruber, P. Johnson, and A. C. Myers. Theta reference manual, preliminary version. Memo 88, Programming Methodology Group, MIT Laboratory for Computer Science, 1995. www.pmg.lcs.mit.edu/Theta.html.
18. Y. Lu, J. Potter, and J. Xue. Validity invariants and effects. In *ECOOP*, volume 4609 of *LNCS*, pages 202–226. Springer, 2007.
19. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.
20. P. Müller and A. Rudich. Ownership transfer in Universe Types. In *OOPSLA*, volume 42, number 10 in *SIGPLAN Notices*, pages 461–478. ACM, 2007.
21. D. A. Naumann. Observational purity and encapsulation. *TCS*, 376(3):205–224, 2007.
22. J. Östlund, T. Wrigstad, D. Clarke, and B. Åkerblom. Ownership, uniqueness and immutability. In *IWACO*, 2007.
23. M. S. Tschantz and M. D. Ernst. Javari: adding reference immutability to Java. In *OOPSLA*, volume 40, number 10 in *SIGPLAN Notices*, pages 211–230. ACM, 2005.
24. Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kieżun, and M. D. Ernst. Object and reference immutability using java generics. In *ESEC-FSE*, pages 75–84. ACM, 2007.