Object Ownership in Program Verification

Werner Dietl[1] and Peter Müller[2]

[1] University of Washington
wmdietl@cs.washington.edu
[2] ETH Zurich
peter.mueller@inf.ethz.ch

Abstract. Dealing with aliasing is one of the key challenges for the verification of imperative programs. For instance, aliases make it difficult to determine which abstractions are potentially affected by a heap update and to determine which locks need to be acquired to avoid data races. Object ownership was one of the first approaches that allowed programmers to control aliasing and to restrict the operations that can be applied to a reference. It thus enabled sound, modular, and automatic verification of heap-manipulating programs. In this paper, we present two ownership systems that have been designed specifically to support program verification—Universe Types and Spec#'s Dynamic Ownership—and explain their applications in program verification, illustrated through a series of Spec# examples.

1 Introduction

Dealing with aliasing is one of the key challenges for the verification of imperative programs. To understand some of the difficulties caused by aliasing, consider an implementation of a list consisting of a list head and a linked node structure, and two list instances `l1` and `l2`. Typical verification tasks include:

1. *Framing*: Does a call to `l1.Add` affect properties of `l2` such as `l2`'s length?
2. *Invariants*: Does a call to `l1.Add` possibly break the invariant of `l2`?
3. *Locking*: If each list method acquires the list head's lock before performing a list operation, can there be data races on the list structure?

The answers to these questions depend on aliasing. If `l1` and `l2` have disjoint node structures then the answer to all three questions is "no". However, if they possibly share the nodes then the answer might be "yes":

1. Appending a new node to the shared node structure affects `l2`'s length if the length is computed by traversing the nodes until a null reference is reached.
2. Appending a node might also break `l2`'s invariant, for instance if `l2` contains a `last` field and the invariant `last.next==null`.
3. When two threads acquire the locks of `l1` and `l2`, respectively, then they might update the shared node structure concurrently, leading to a data race.

This example shows that program verifiers need information about aliasing to decide when properties are preserved, which invariants to check, or which locks to require for a heap access, to mention just some of the most common verification tasks.

Early work on verifying heap-manipulating programs provided only partial solutions to the problems caused by aliasing. Some techniques use an explicit heap representation and require the user to reason about the consequences of each heap update explicitly [59,28], which compromises abstraction and information hiding. Moreover, the resulting proof obligations are non-modular and difficult to prove automatically. Leino and Nelson [34,43] addressed the abstraction problem by allowing a specification to provide information about the footprint of a heap property without revealing the property itself; however, the resulting proof obligations make heavy use of reachability predicates and are, thus, difficult to discharge automatically. Yet another approach is to make unsound assumptions about the effects of heap updates and to optimize the proof obligation to strike a good balance between the errors that can be detected on one side and the annotation overhead, modularity, and automation on the other side [24]. So towards the end of the last millennium, there was no verification technique for heap-manipulating programs that was sound, modular, and amenable to automation.

This situation changed with the invention of object ownership and ownership types [11]. Ownership provides two important benefits for program verification. First, it allows programmers to describe the *topology* of heap data structures in a simple and natural way, at least for hierarchical data structures. For instance, ownership can express that two lists have disjoint node structures, without resorting to reachability predicates. Ownership types provide an automatic way of checking that an implementation conforms to the intended topology. Hierarchical topologies help for instance with proving data race freedom. Second, ownership can be used to define and enforce *encapsulation disciplines*, which describe what references may exist in a program execution and which operations may be performed on these references. For instance, an encapsulation discipline may allow arbitrary objects (such as iterators) to read the nodes of a list, but only allow the list header and its nodes to modify the node structure. Restricting write accesses helps for instance with verifying object invariants.

In this paper, we summarize the topology and encapsulation disciplines that are used in ownership-based program verification (Sec. 2) and present two ways of enforcing them—a type system called Universe Types [49,22,18,20] and a verification methodology called Dynamic Ownership [37] (Sec. 3). The main part of the paper surveys applications of object ownership in program verification. We discuss how the ownership topology is used for effect specifications, framing, proving termination, and for defining the semantics of object invariants (Sec. 4). Then we show how encapsulation disciplines on top of ownership systems are used to verify object invariants, to define and check object immutability, and to prove the absence of data races (Sec. 5). For each of these verification tasks, we describe the problem, explain the ownership solution, and briefly summarize

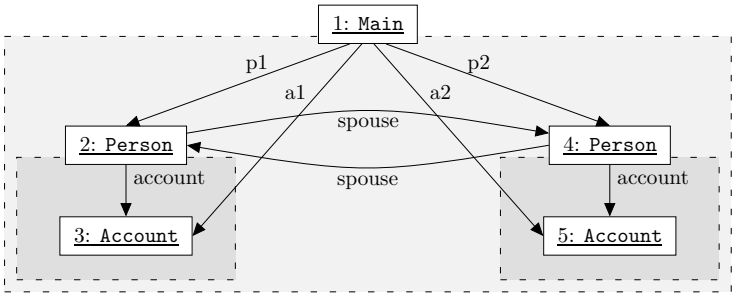

Fig. 1: Example of an ownership structure. Boxes and arrows depict objects and references, respectively. Dashed boxes enclose the objects of an ownership context; the owner of these objects sits atop the dashed box.

alternative solutions. We illustrate the ownership solutions using examples written and verified in the Spec# system [3,41]. The Spec# examples as well as a translation to Java with JML [33] and Universe annotations are available from `www.pm.inf.ethz.ch/publications/OwnershipInVerification/`.

2 Ownership Topology and Encapsulation

This section gives a brief overview of the ownership topology and the encapsulation disciplines that have been used to support program verification.

Topology. The *ownership topology* provides a hierarchical structure of the objects in the heap. Each object is *owned* by at most one other object, its *owner*. The ownership relation is acyclic, that is, the ownership topology is a forest of ownership trees, and objects without owner are roots. The set of objects with the same owner is called an *ownership context*.

Fig. 1 illustrates an ownership topology, where object 1 of class `Main` owns the `Person` objects 2 and 4. Each of the `Person` objects owns an `Account` object. Since `Person` objects own their `Account` objects, the ownership topology guarantees that `Account` objects are never shared among `Person` objects.

Some ownership systems allow the ownership topology to change during the execution of a program by supporting ownership transfer [12,37,53]. For simplicity, we ignore ownership transfer in this paper; once an object has been assigned an owner, the owner does not change for the remaining lifetime of the object.

Encapsulation. The ownership topology can be used to define various encapsulation disciplines that restrict references or interactions between objects. For the verification of imperative programs, reasoning about side effects is one of the key challenges. Therefore, the most widely used encapsulation discipline in ownership-based verification is the owner-as-modifier discipline, which restricts

the heap modifications that a method may perform. An alternative encapsulation discipline is the owner-as-dominator discipline, which restricts the existence of references.

The *owner-as-modifier* discipline [22,18] enforces that all modifications of an object are initiated by the object's owner. More precisely, when a method modifies an object o then for each (transitive) owner o' of o, the call stack contains a method execution with o' as receiver. For instance, whenever a method modifies the `Account` object 3 in Fig. 1 then there must be method executions in progress with objects 2 and 1 as receivers.

The owner-as-modifier discipline allows an object to control modifications of the objects it (transitively) owns. In particular, the owner can prevent certain modifications or maintain properties of the objects it owns. For instance, the only way for `Person` object 4 to modify `Account` object 3 is to invoke a method on object 3's owner (object 2), which can then impose appropriate checks, for instance, to maintain its own invariant.

In order to enforce the owner-as-modifier discipline modularly, a checker needs to know which methods potentially modify the heap and which methods do not: calls to methods with side effects (*impure* methods) must be restricted, whereas side-effect free (*pure*) methods may be called on any object. For this purpose, we assume that side-effect free methods are labelled as pure. How to check method purity is beyond the scope of this paper [54,63].

The *owner-as-dominator* discipline [11] enforces that the owner o' of an object o is a dominator for every access path from a root of the ownership forest to o. For instance, all access paths from object 1 to object 3 must pass through object 3's owner, object 2. The references `a1` and `a2` that bypass the owning `Person` objects are not permitted in this discipline.

The owner-as-dominator discipline allows an object to control the existence of access paths to the objects it (transitively) owns. Therefore, the owner controls read and write access to these objects, which is for instance useful to enforce locking strategies.

3 Enforcing Topology and Encapsulation

This section describes two alternative approaches to enforce the ownership topology and encapsulation discipline: a type system called Universe Types and a verification methodology called Dynamic Ownership.

3.1 Universe Types

Universe Types [20,21,19,22] is a static type system that enforces the ownership topology and the owner-as-modifier encapsulation discipline as described in the previous section.

Ownership Qualifiers. The type system equips each occurrence of a reference type with one of five type qualifiers[3]. These *ownership qualifiers* describe the position of an object in the ownership hierarchy relatively to the current object `this`:

- `peer` indicates that an object has the same owner as the current object.
- `rep` (short for "representation") indicates that an object is owned by the current object.
- `any` indicates that no static knowledge about an object's owner is available, and the owner is arbitrary.
- `lost` indicates that no static knowledge about an object's owner is available, but there are constraints on the owner.
- `self` indicates that an object is the current object.

The example in Fig. 2 (left column) illustrates the use of ownership qualifiers. A `Person` object has a reference to an `Account`, which is owned by the `Person` object; therefore, the `account` field has a `rep` qualifier. A `Person` object and its spouse share the same owner; therefore, the `spouse` field has a `peer` qualifier. Instances of class `Main` own two `Person` objects as indicated by the `rep` qualifier for the fields `p1` and `p2`.

In Universe Types, the owner of an object is determined when the object is created. The owner is indicated by a `rep` or `peer` qualifier in the `new` expression, as illustrated by method `Demo`.

Viewpoint Adaptation. The ownership qualifier of a type depends on the current object `this`. Therefore, when the current object changes, the ownership qualifiers need to change as well. We call this process *viewpoint adaptation*. As a simple example, consider class `Person` from Fig. 2. Field `spouse` has declared type `peer Person`, expressing that the current `Person` instance and the referenced person share the same owner. However, for any `Person` reference `p`, the type of `p.spouse` is not necessarily the declared type `peer Person`. For instance, when `p` is of type `rep Person` then `p` is owned by the current object `this`. Since `p` and `p.spouse` have the same owner, `p.spouse` is also owned by `this` and, therefore, has type `rep Person`. That is, the declared type of `spouse` was adapted to the new viewpoint, `this`.

More formally, we define viewpoint adaptation of declared qualifier u' from the viewpoint given by qualifier u to the current object `this` using function $\triangleright$, which is defined as follows:

$$
\begin{array}{lll}
\texttt{self} \triangleright u' & = u' & \\
\texttt{peer} \triangleright \texttt{peer} & = \texttt{peer} & \\
\texttt{rep} \triangleright \texttt{peer} & = \texttt{rep} & \\
u \triangleright \texttt{any} & = \texttt{any} & \\
u \triangleright u' & = \texttt{lost} & \text{otherwise}
\end{array}
$$

[3] Arrays and instantiations of generic types may have more than one qualifier.

Universe Types:

```
class Account {
  int value;
}

class Person {
  rep Account account;
  peer Person spouse;

  Person() {
    account = new rep Account();
    spouse = null;
  }
}

class Main {
  rep Person p1;
  rep Person p2;

  void Demo() {
    p1 = new rep Person();
    p2 = new rep Person();


    p1.spouse = p2;
    p2.spouse = p1;


    any Account a1 = p1.account;
    any Account a2 = p2.account;
    int total = a1.value + a2.value;

    // forbidden by Topology
    p1.account = p2.account;
    // forbidden by Encapsulation
    a1.value += 10000;
  }
}
```

Dynamic Ownership:

```
class Account {
  int value;
}

class Person {
  [Rep] Account account;
  [Peer] Person spouse;

  Person() {
    account = new Account();
    spouse = null;
  }
}

class Main {
  [Rep] Person p1;
  [Rep] Person p2;

  void Demo() {
    p1 = new Person();
    p2 = new Person();

    expose(this) {
      p1.spouse = p2;
      p2.spouse = p1;
    }

    Account a1 = p1.account;
    Account a2 = p2.account;
    int total = a1.value + a2.value;

    // forbidden by Topology
    p1.account = p2.account;
    // forbidden by Encapsulation
    a1.value += 10000;
  }
}
```

Fig. 2: Illustration of the ownership topology and encapsulation. The code on the left uses Universe Types; the code on the right uses Spec#'s Dynamic Ownership. We omit accessibility qualifiers and non-null annotations for brevity. Execution of method `Main.Demo` results in the ownership structure from Fig. 1.

Let us again consider class `Person` from Fig. 2 where field `spouse` has the declared type `peer Person`. For a reference `p` of type `rep Person`, to determine the type of a field access `p.spouse`, one adapts the declared qualifier (`peer`) from the qualifier of the receiver (`rep`) to `this`; that is, we determine that `rep` ▷ `peer` is `rep` and the resulting type is therefore `rep Person`. This corresponds to the intuition that the field access first goes into the representation and then stays within the current representation—resulting in a reference that points into the current representation.

Viewpoint adaptation is applied whenever a declared type needs to be adapted to the current object, that is, for field accesses and updates (adapting the declared field type), method calls (adapting the declared method parameter and return

types), and object instantiations (adapting the declared constructor parameter types). For generic types, it is also applied to the upper bounds of type parameters.

There is an important distinction between the ownership qualifiers `any` and `lost`. Qualifier `any` expresses that there is no static constraint on the ownership of the referenced object. In our example, local variable `a1` can be used to reference an arbitrary account. On the other hand, `lost` expresses that there exists an ownership constraint, but this constraint is not expressible in the type system. In method `Demo`, the access `p1.account` first follows the rep reference `p1` and then another rep reference `account`. There is no static qualifier to express the resulting ownership relationship. Viewpoint adaptation yields the `lost` qualifier for the type of `p1.account`. Such a reference can be used for reading, but an update of `p1.account` is forbidden, as the type system cannot statically ensure that the assigned value satisfies the ownership constraint expressed by the `rep` qualifier in the declaration of `account`. For the same reason, method invocations and object instantiations are forbidden if the parameter types after viewpoint adaptation contain `lost`. In that case, the method implementation expects a specific owner, but the type system cannot guarantee that the caller passes an argument with the expected owner because the ownership information has been lost during viewpoint adaptation.

Subtyping. The ownership qualifiers are in a simple ordered relationship. The `self` qualifier is used only for the current object `this`, which is evidently a peer of itself; therefore `self` is a subtype of `peer`. Both `peer` and `rep` are subtypes of `lost` as the former are more precise than the latter. Finally, all ownership qualifiers are subtypes of `any`, because the latter does not express any ownership information.

In method `Demo`, we can assign `p1.account` to local variable `a1` because the qualifier of the right-hand side, `lost`, is a subtype of `a1`'s qualifier, `any`.

Encapsulation. Universe Types optionally enforce the owner-as-modifier encapsulation discipline. Field updates and calls to impure methods require that the ownership qualifier of the receiver is neither `any` nor `lost`. Field reads and calls to pure methods are permitted for arbitrary receivers.

Consider the last assignment `a1.value += 10000` in method `Demo`. This assignment does not violate the ownership topology, because field `value` is of a primitive type. However, it does violate the owner-as-modifier discipline. Since the receiver `a1` is of type `any Account`, we have no static knowledge of its owner. Therefore, the type system cannot ensure that the owner controls this modification and, thus, it is forbidden.

3.2 Dynamic Ownership

Dynamic Ownership [37] enforces the ownership topology and an encapsulation discipline similar to owner-as-modifier through program verification. To support inheritance, Dynamic Ownership uses object-class pairs as owners. Here, we

present a simplified version without the class-component. Dynamic Ownership has been implemented in the Spec# language, and we will be using Spec# in the rest of the paper.

Ownership Attributes. To encode the ownership relation, Dynamic Ownership adds a ghost field `owner` to each object. The ownership topology is then expressed through specifications over this ghost field. In particular, Spec# offers two attributes (annotations in Java terminology) `[Peer]` and `[Rep]` that can be added to field declarations[4]. Declaring a field f with a `[Peer]` attribute is similar to the following object invariant, but is enforced differently as we explain below:

```
invariant f != null ==> f.owner == this.owner;
```

Analogously, a `[Rep]` attribute is similar to the following object invariant:

```
invariant f != null ==> f.owner == this;
```

The example in Fig. 2 (right column) illustrates the use of ownership attributes. The fields `account`, `spouse`, `p1`, and `p2` are declared with attributes analogous to the ownership qualifiers in Universe Types. In contrast to Universe Types, the declarations of methods and local variables do not include ownership information. For methods, ownership information can be expressed via pre- and postconditions; for locals, the verifier keeps track of any information about the `owner` field.

In contrast to Universe Types, Spec# does not determine the owner of an object when it is created. Fresh objects start out as un-owned. The owner is set when an un-owned object is assigned to a rep or peer field of another object. More precisely, for a peer field f and the assignment $o.f = e$ the verifier imposes a proof obligation that either e is null, that o and e have the same owner, or that e is un-owned. In the latter case, it sets e's owner to o's owner. If both o and e are un-owned, they stay un-owned, but the verifier records that the objects are peers (we say they form a *peer group*). When one object of a peer group gets its owner assigned then the owner of all objects of the peer group is set to ensure that they stay peers of each other. The treatment of assignments to rep fields is analogous, but does not create peer groups.

Method `Demo` in Fig. 2 illustrates this process. After creating the first `Person` object, it is initially un-owned. Its owner is set to `this` when it gets assigned to field `p1`, because `p1` is a rep field. The assignment `p1.account = p2.account` towards the end of the method fails to verify since reference `p2.account` is not known to be null and the owner of the object `p2.account` is `p2`, because field `account` is a rep field. That is, the right-hand side is neither un-owned nor owned by `p1`, making the assignment to a rep field of `p1` invalid.

Spec#'s semantics of field updates guarantees that rep and peer fields satisfy the corresponding ownership invariants in all execution states, similar to a type system. However, Dynamic Ownership permits a more flexible initialization of an object's owner than most ownership type systems, because an object's owner need not be fixed when the object is created. Like Universe Types, Spec# does

[4] Additional attributes allow one to handle arrays and instantiations of generic types.

not allow the owner of an object to change after it has been set, but supporting ownership transfer is possible [37].

Encapsulation. Spec# enforces an encapsulation discipline that controls modifications of objects, similar to the owner-as-modifier discipline. In Spec#, each object is either in the *valid* or in the *mutable* state. We say that an object is *consistent* if it is valid and its owner (if any) is mutable; it is *peer consistent* if the object and all of its peers are consistent.

Valid objects must satisfy their object invariants, whereas mutable objects may violate their object invariants (but not the ownership invariants mentioned above). A proof obligation for field updates enforces that the receiver is mutable or—in case the update is known not to violate the receiver's object invariant—consistent.

Fresh objects are initially mutable and become valid when the `new` expression terminates. Before a valid object can be modified, it has to be *exposed*. For this purpose, Spec# provides a block statement `expose(`o`) { ... }` with the following semantics: The statement first asserts that o is consistent. Then o becomes mutable and the body of the block statement is executed. Finally, the `expose` statement asserts that o's object invariant holds (see Sec. 5.1) and that all objects owned by o are valid, and then makes o valid.

The Spec# methodology guarantees that in each execution state all objects (transitively) owned by a valid object are also valid. Consequently, before updating an object o, all of o's transitive owners need to be exposed. Similarly to the owner-as-modifier discipline, this gives the owners the possibility to control modifications, for instance, to maintain invariants over the owned objects.

By default, impure methods (and constructors) in Spec# require their receiver and arguments to be peer consistent and ensure that their result is also peer consistent. This allows the method to expose the receiver and the arguments and modify their state. Pure methods by default require the receiver and arguments (and their peers) to be valid, and ensure peer validity of the result.

In method `Demo` (Fig. 2, right column), these defaults imply that the receiver `this` is consistent at the beginning of the method. The newly created `Person` objects are consistent when their `new`-expressions terminate; after assigning them to the fields `p1` and `p2`, they remain valid but are no longer consistent because they now have a valid owner, `this`. Therefore, before updating their `spouse` fields, the method has to expose `this` to make the `Person` objects consistent, which is sufficient to permit the update since class `Person` does not declare an object invariant. (If it did, we would also have to expose `p1` and `p2`.) The fact that we can read the `account` fields as well as the `value` fields of the `Account` objects without exposing any objects illustrates that Spec#'s encapsulation discipline controls modifications, but not read access. The last statement fails to verify because the object `a1` is neither mutable nor consistent. In order to verify the assignment, the method would have to ensure that all of `a1`'s transitive owners are mutable. By the default precondition of method `Demo`, the verifier knows that the transitive owners of `this` are mutable. So it suffices to expose `this` and `p1`:

```
expose(this) {
```

```
  expose(p1) {
    a1.value += 10000;
  }
}
```

This example illustrates that Spec#'s encapsulation discipline is slightly more flexible than owner-as-modifier. For the update of `a1.value`, the owner-as-modifier discipline requires that on each of `a1`'s transitive owners, a method execution is currently on the stack. By contrast, Spec# requires only that these owners are mutable, that is, that an `expose` block is currently being executed. However, these `expose` blocks may occur in any method execution, not necessarily in methods whose receiver is the owner. Here, we do not have to call a method on `p1` in order to update `a1.value`.

4 Using Ownership Topologies in Verification

Ownership has numerous applications in program verification. In this section, we present four ways to use the hierarchical heap topology of ownership systems. Applications that in addition rely on encapsulation are presented in Sec. 5. We illustrate each application by an implementation of an array list, written and verified in Spec# [3,41]. The Spec# examples as well as a translation to Java with JML [33] and Universe annotations are available from the companion website. Fig. 3 shows a simple list interface; class `ArrayList` in Fig. 4 implements this interface.

```
interface List {
  void Add(object! e);
    ensures Contains(e);
    ensures Length() == old(Length()) + 1;

  void Remove(object! e);
    ensures old(Contains(e)) ==> Length() == old(Length()) - 1;

  [Pure] bool Contains(object! e);

  [Pure] int Length();
}
```

Fig. 3: A simple list interface. In Spec#, an exclamation point after a reference type indicates a non-null type. Method pre- and postconditions are written as requires and ensures clauses, respectively. The `old` keyword in postconditions is used to refer to the prestate value of an expression. The attribute `Pure` indicates that a method does not modify any objects that are allocated in its prestate; pure methods may be used in specifications because they are side-effect free.

```
public class ArrayList : List {
  [Rep][SpecPublic] object[]! elems = new object[32];
  int next;

  invariant 0 <= next && next <= elems.Length;
  invariant elems.GetType() == typeof(object[]);
  invariant forall{int i in (0: next); elems[i] != null};

  public ArrayList()
    ensures Length() == 0;
  { }

  public void Add(object! e)
  {
    expose(this)
    {
      if(next == elems.Length) // resize the array
      {
        object[] tmp = new object[elems.Length * 2 + 1];
        Array.Copy(elems, tmp, next);
        elems = tmp;
      }
      elems[next] = e;
      next++;
    }
    assert elems[next-1] == e;
  }

  [Pure] public bool Contains(object! e)
    ensures result == exists{int i in (0: next); elems[i] == e};
  {
    for(int i = 0; i < next; i++)
      invariant forall{int j in (0: i); elems[j] != e};
    {
      if(elems[i] == e)
        return true;
    }
    return false;
  }

  // other methods omitted
}
```

Fig. 4: An array list implementation of the `List` interface. The methods inherit the contracts defined in `List`. The `invariant` clauses after the field declarations declare object invariants, whereas the `invariant` clause in method `Contains` declares a loop invariant. The `forall` and `exists` expressions quantify over half-open integer intervals. The `SpecPublic` attribute on `elems` allows the field to appear in specifications of public methods.

4.1 Effect Specifications

Verification of imperative programs relies heavily on effect specifications to characterize the parts of the heap that a method may read, modify, de-allocate, lock, etc. [26]. For example, `List`'s `Add` method may modify the internal representation of the list (for instance, an underlying array), but nothing else.

Problem. A naive approach to specifying effects is to enumerate the affected heap locations in the effect specification. For instance, one could specify the write effect of method `Add` in class `ArrayList` using the following specification clause:

```
modifies elems, next, elems[*];
```

which gives permission to modify the list's `elems` and `next` fields as well as all elements of the `elems` array. The problem with this approach is that it violates information hiding, which has several negative consequences: (1) Changes of the internals of a data structure become visible to clients and, thus, might require re-verification of client code. (2) Enumerating locations is not possible for interfaces, which do not have a concrete implementation. (3) It is difficult to allow overriding methods in subclasses to modify the additional fields declared in subclasses (this is sometimes called the extended state problem [35]).

When the effect of a method includes only fields of its parameters then these problems can be solved by using wildcards or static data groups [35]. For instance, Spec# provides the wildcard syntax o`.*` to denote all fields of an object o. However, this solution does not apply to effects that include objects reachable from the method parameters such as the array object in `this.elems`. In this case, one needs to specify a path to the affected objects (here, `elems`), which reveals implementation details. This is particularly problematic when a method affects an unbounded number of locations, for instance, modifies fields of all nodes in a linked list; it is then not possible to enumerate all of these locations statically. So the problem is:

How to express implementation-independent effect specifications?

Ownership Solution. Clarke and Drossopoulou [10] showed that the hierarchical heap topology of ownership systems provides a natural abstraction mechanism for effects by using a reference to an object o to represent o and all objects (transitively) owned by o (that is, the ownership tree rooted in o).

This idea is adopted by Spec# to specify read and write effects. Unless indicated otherwise, every non-pure method in Spec# has the default write effect

```
modifies this.*;
```

which allows the method to modify all fields of the receiver object (by the wildcard) as well as all fields of all objects (transitively) owned by the receiver (by the abstraction mechanism mentioned above). This effect specification does not reveal any implementation details. In our example, it applies to the `Add` method

of interface `List` and of class `ArrayList`. Note that the latter satisfies the effect specification because it modifies at most the following locations: (1) fields of the receiver, which is permitted by the wildcard in the effect specification, (2) the array object in `this.elems`, which is permitted because the array is owned by the receiver, and (3) elements of the new array it allocates; the initialization of fresh objects is generally not considered a write effect and, thus, does not have to be included in the effect specification. Note that in the implementation of Fig. 4, this write effect does not allow the `Add` method to modify the states of the objects stored in the `elems` array because they are not owned by the receiver. Spec# provides an `[ElementsRep]` attribute for arrays to declare such an ownership relation if desired.

Similarly, the default read effect of pure methods in Spec# contains all fields of the receiver as well as all fields of all objects (transitively) owned by the receiver. Other read effects can be specified using attributes on the method declaration. The default allows methods `Contains` and `Length` of class `ArrayList` to read `next`, `elems`, and the elements of `elems`.

Spec# checks write effects by generating appropriate proof obligations [2]. A crucial property of these proof obligations is that they avoid reachability predicates (such as transitive ownership) because these are not handled well by automatic theorem provers. Read effects are checked syntactically, but a more precise checking through verification is also possible [39].

Other Solutions. Leino proposed a formalism based on explicit dependencies between model fields and concrete fields to provide implementation-independent specifications of effects [34]. When an effect includes a model field then it also includes all fields the model field (transitively) depends on. Drawbacks of this approach are that it uses reachability predicates (transitive dependencies) and that its soundness argument is complicated [43]. A similar approach is used in dynamic data groups [44], but with a simpler formalism. The second author's thesis [49] combined explicit dependencies with ownership.

Kassios developed a powerful technique to specify effects called Dynamic Frames [32]. A dynamic frame is a set of objects or locations. Effect specifications simply mention such sets; clients reason in terms of these sets and their disjointness, but do not need to know the set contents, which preserves information hiding. In Kassios' work, dynamic frames are encoded as functions of the heap. Later work, for instance on Dafny [36] and Region Logic [1] encodes dynamic frames using ghost variables to increase automation with SMT solvers.

Another approach to specifying effects is via permissions [9]. Each memory location is associated with a permission, and a method may access a location only if it has the permission to do so. Read and write effects can then be inferred from permission specifications. This approach is for instance taken in Separation Logic [5,60] and Implicit Dynamic Frames [64,40]. Abstraction is expressed via abstract predicates [58]. Both Dynamic Frames and permission systems require a higher annotation overhead than the ownership solution, but have the big advantage that they are not limited to hierarchical structures.

4.2 Framing

One of the key challenges in verifying imperative programs is to determine whether the heap modifications performed by a piece of code affect the validity of heap properties. The client class in Fig. 5 illustrates this problem. The property `l1.Contains("one")` is established by the call `l1.Add("one")`. To prove that it still holds at the end of method `Frame` requires framing. One has to prove that the call to `l2.Remove` does not invalidate the property, which could for instance happen if `l1` and `l2` shared the same array.

```
class Client {
  public static void Frame(List! l1, List! l2)
    requires l1 != l2;
    modifies l1.*, l2.*;
  {
    l1.Add("one");
    l2.Remove("one");
    assert l1.Contains("one");
  }
}
```

Fig. 5: A client of the `List` interface. Proving the assertion requires framing.

Problem. Framing is based on effect specifications. If the write effect of a piece of code is disjoint from the read effect of a heap property then executing the code does not affect the validity of the property. Framing is trivial with the naive effect specifications, where the affected locations are explicitly enumerated. However, when effect specifications use abstraction then clients must be able to determine whether two effects are disjoint without actually knowing the concrete set of affected locations. So the problem is:

How to determine whether two effects overlap?

Ownership Solution. As we discussed in Sec. 4.1, ownership-based effect specifications use the root of an ownership tree to abstract over the locations of all objects in the tree. Because each object has at most one owner, the trees rooted in objects o and p are either disjoint or one tree is a (not necessarily proper) sub-tree of the other. So to prove that the effects characterized by o and p are disjoint, it is sufficient to show that $o \neq p$ and neither o (transitively) owns p nor vice versa.

In Spec#, this property is proved as follows. As we have explained in Sec. 3.2, impure methods in Spec# require by default that the receiver and each explicit argument are consistent, that is, that they are valid and their owners (if any) are

mutable. Since Spec# guarantees that all objects (transitively) owned by a valid object are also valid, this default requirement implies that it is not possible for one argument to own another. Thus, any two arguments either refer to the same object or to disjoint ownership trees. In the example from Fig. 5, the precondition guarantees that `l1` and `l2` point to different objects and, thus, the trees rooted in `l1` and `l2` are disjoint, which is sufficient to prove that `l2.Remove` does not affect the validity of `l1.Contains("one")`.

Other Solutions. The three techniques for effect specifications discussed in Sec. 4.1 solve the framing problem in different ways.

Techniques that use explicit dependencies without ownership include rules that allow one to derive the absence of certain dependencies. If we know that a model field a does *not* (transitively) depend on another field b then some code with write effect a will not affect a property with read effect b. Reasoning about such specifications again involves reachability. Dynamic data groups simplify this reasoning by imposing various restrictions on dependencies. For instance in our example, the *pivot uniqueness* requirement guarantees that the lists `l1` and `l2` operate on different arrays.

With Dynamic Frames, specifications have to state the disjointness of frames explicitly. A common idiom is to put all locations that belong to a data structure into one frame `state`. Methods of the data structure then typically read or write `state`. With this effect specification, method `Frame` in Fig. 5 requires the additional precondition $\texttt{l1.state} \cap \texttt{l2.state} = \emptyset$.

Permission-based systems use two ingredients for framing. First, they require every heap property (in particular, every abstract predicate) to be *self-framing*, that is, to include permissions for all heap accesses in the property. Second, they support a *separating conjunction* $*$, which holds if both conjuncts hold and their permissions are disjoint. So if formulas P and Q hold separately, and a piece of code operates using the permissions specified in P then it is guaranteed not to affect Q. In our example, let's assume an abstract predicate `valid` that contains all permissions for the list data structure and that is used as pre- and postcondition of methods `Add`, `Remove`, and `Contains`. Method `Frame` in Fig. 5 can then be verified using the additional precondition $\texttt{l1.valid} * \texttt{l2.valid}$, which implies that `l2.Remove`'s write effect is disjoint from `l1.Contains`'s read effect.

4.3 Termination

Program verification often includes termination proofs for two reasons. First, termination is a desired property of many implementations. Second, many specification formalisms support recursive functions or pure methods, which are encoded for the prover as uninterpreted function symbols with appropriate axioms. To ensure the consistency of these axioms, one needs to show that the recursion is well founded [62].

Problem. The standard approach to termination proofs is to find a ranking function that maps the program state to a well-founded set such as the natural numbers, and to prove that each recursive call or loop iteration decreases the value of the ranking function [15]. Modular verifiers that check termination require a ranking function for each loop and recursive method, which is difficult to specify for loops and methods that traverse object structures. The example in Fig. 6 illustrates the problem. The recursive call in method `GetHashCode` should be permitted, provided that it does not eventually call `GetHashCode` on the list. This could for instance happen if the hash code of an array was defined in terms of the hash codes of its elements and if the list was stored in its own array.

```
[Pure] public override int GetHashCode()
  ensures result == elems.GetHashCode();
{ return elems.GetHashCode(); }
```

Fig. 6: `GetHashCode` of class `ArrayList`. Spec# proves that recursive specifications of pure methods are well founded. Here, the height of the receiver in the ownership hierarchy is implicitly used as ranking function.

In general, one needs to show that a recursive heap traversal does not go in cycles, which is typically done by using the distance from the end of the traversal as a ranking function. However, denoting this distance in specifications requires a reachability predicate (for instance, this list node reaches the end of the list in n steps), which are notoriously difficult to handle for automatic verifiers based on SMT solvers. A work-around is to encode the distance using ghost state; for instance, each list node stores its distance from the end of the list. However, maintaining the ghost state increases the specification overhead. Moreover, adding ghost state to library classes and arrays is not possible. So the problem is:

How to specify ranking functions for heap traversals?

Ownership Solution. Since the ownership topology is acyclic, each recursion that traverses an ownership tree only downwards or only upwards is well founded. In other words, for traversals towards the leaves, we can use the height of an object in the ownership tree as a ranking function, provided that the tree does not grow during the traversal. Analogously, we can use the depth in the tree for traversals towards the root. This approach does not lead to the annotation overhead incurred by solutions based on ghost state. Moreover, it can be checked syntactically, which avoids reasoning about reachability predicates. For downward traversals, it is sufficient to check that the receiver of each recursive call is of a `rep` type. One could use an analogous check for upward traversals, if the ownership system provided an "owner" type qualifier or attribute, which is not the case in Spec#.

Spec# uses a lexicographic ordering as ranking function [17]. The first component is the height of an object in the ownership tree. Since termination is proved only for pure (side-effect free) methods, the tree cannot grow during the traversal. The second component is a static ordering of all method declarations in a program; it is determined by the order in which the declarations occur in the program text and can also be specified explicitly via attributes on method declarations. The example in Fig. 6 is accepted by Spec# because the receiver of the recursive call, `elems`, is declared with the `[Rep]` attribute.

Other Solutions. Another ranking function for recursive methods is the size of their read effects [36]. If the read effect gets strictly smaller with each recursive call then the recursion will terminate eventually. For effect specifications based on dynamic frames, this property can be verified by comparing set cardinalities (if the prover supports them) or by checking set inclusion. The solution based on read effects has the advantage that it does not restrict the receivers of recursive calls, which is sometimes useful. On the other hand, the ownership-based solution works even if the read effect does not get smaller. For instance, the example in Fig. 6 would verify even if the `GetHashCode` method of `object` and of `ArrayList` were permitted to read the whole heap.

For effect specifications based on permissions, one can check that after passing the required permissions to the recursive call, a non-empty permission set remains with the caller. This solution works well for side-effect free methods, similar to the ownership-based solution. For methods that create new objects, it is difficult to make the checks sound and nevertheless permit calls on newly-created objects.

4.4 Multi-Object Invariants

Consistency criteria of data structures are often specified as object invariants [23,48]. Non-trivial object invariants do not hold in all execution states; for instance, the invariant of a fresh object typically does not hold until the object is initialized, and some methods violate and later re-establish an invariant when they update a data structure. Therefore, any specification language that supports object invariants must define when invariants are expected to hold.

Problem. A common semantics for object invariants is to require the invariants of all allocated objects to hold in the pre- and poststates of each method call [27,48,59]. This *visible state semantics* is suitable for object invariants that constrain the state of a single object[5], but it is too restrictive for *multi-object invariants*, which relate the states of multiple objects. The problem is illustrated by class `Map` in Fig. 7.

Class `Map` implements a map; the keys and values are stored in two lists. Consequently, `Map` maintains an invariant that these lists have the same length.

[5] Visible state semantics is also problematic in the presence of call-backs [2,23], but we ignore this issue here.

```
class Map {
  [Rep][SpecPublic] List! keys = new ArrayList();
  [Rep] List! values = new ArrayList();

  invariant keys.Length() == values.Length();

  public void Put(object! key, object! value)
    requires !keys.Contains(key);
    requires Owner.Different(key, this);
    requires Owner.Different(value, this);
  {
    expose(this) {
      keys.Add(key);
      values.Add(value);
    }
  }

  // other methods omitted
}
```

Fig. 7: A map implementation that stores the keys and values in two lists. The invariant relates the states of both lists. The second and third precondition of `Put` are necessary to show that `keys` and `values` are peer consistent after exposing `this`, which is required to satisfy the default precondition of `List.Add` (see Sec. 3.2).

This invariant is a multi-object invariant since if relates the states of a `Map` object and of the two `List` objects referenced by the `Map` object. Method `Put` adds a new key-value pair to the map by adding the key and the value to the respective lists. The call to `keys.Add` violates the `Map` invariant by increasing the length of one list. The subsequent call to `values.Add` re-establishes the invariant. However, note that the invariant does not hold in the poststate of the first call and in the prestate of the second call, which are visible states. This example shows that the visible state semantics is too restrictive to handle useful multi-object invariants. So the problem is:

How to define a semantics for multi-object invariants?

Ownership Solution. In ownership systems, owned objects form the internal representation of their owner, for instance, the `List` objects in our example form the internal representation of the `Map` object that owns them. Consequently, modifications of owned objects should be seen as internal operations of the owner, which may violate the owner's invariant as long as it gets re-established when the modifications are completed. This is analogous to allowing field updates to violate a single-object invariant as long as it gets re-established before the enclosing method terminates.

Based on this idea, we can use the ownership topology to refine the standard visible state semantics to the following *relevant invariant semantics* [49,52]. Consider an arbitrary execution of a method m on receiver o. In the pre- and poststate of this execution, the invariants of those allocated objects have to hold that are directly or transitively owned by the owner of o. In other words, m may assume and has to preserve the invariants of o, o's peers, and all objects (transitively) owned by o and its peers. These objects are called the *relevant objects* for o. However, m must not assume nor has to preserve the invariants of o's transitive owners. In the `Map` example, this semantics allows the calls to `keys.Add` and `values.Add` to break the invariant of the `Map` object because it owns the receivers of these calls and, thus, is not relevant for the receivers.

With the relevant invariant semantics, it is generally not possible for an object o to call methods on receivers that are not relevant for o because their invariants are not known to hold. For instance, o cannot call methods on its owner nor on objects in a different ownership tree. To address this limitation, one can allow methods to specify explicitly which invariants they require to hold [37,47]. Such a specification is similar to an effect specification, and we can employ ownership as an abstraction mechanism as discussed in Sec. 4.1. For instance, in Spec# the precondition `p.IsPeerConsistent` requires that all objects relevant for `p` satisfy their invariants. By default, a Spec# method requires that its receiver and all explicit arguments are peer consistent. In our example, the call `keys.Add(key)` requires implicitly that `keys` and `key` are peer consistent. The former follows from the fact that the `Map` object `this` is peer consistent (by the implicit precondition of `Put`) and owns `keys`. The latter follows from the implicit precondition of `Put` that `key` is peer consistent.

Other Solutions. VCC [14] supports two-state invariants that hold between the prestates and poststates of any action, similar to a standard visible state semantics. This semantics is enforced by checking (1) that every action of the program is legal, that is, preserves the invariants of the objects it modifies, and (2) that every invariant is admissible, that is, is reflexive and cannot be broken by any legal action. VCC would reject the invariant of class `Map` because it is not admissible; the action `keys.Add(key)` is legal (it preserves the invariant of the list), but violates the map invariant. To support interesting multi-object invariants, VCC uses ghost state to encode an ownership scheme like the one described above. The key ideas are: (1) to write an invariant I as a dented invariant of the form `valid` $\Rightarrow I$, where `valid` is a boolean ghost field, (2) to add a two-state invariant that the state of an object o does not change while o.`valid` is true, and (3) to relate the `valid` fields of o and its owner using further invariants.

Many specification formalisms do not support object invariants but instead use pre- and postconditions together with model variables [43] or abstract predicates [58] to express consistency criteria.

5 Using Ownership Encapsulation in Verification

The previous section showed that the ownership topology can be utilized in various ways to simplify verification. For some applications, however, the topology alone is not sufficient; it has to be complemented by an encapsulation discipline such as owner-as-dominator or owner-as-modifier to control which objects may be referenced and which operations can be performed on a referenced object. In this section, we present three applications of ownership-based encapsulation to program verification.

5.1 Verifying Multi-Object Invariants

In Sec. 4.4, we presented a semantics for multi-object invariants that defines when invariants are expected to hold, but we did not explain how to verify statically that the expected invariants actually hold. The textbook solution to this problem—assuming invariants to hold in each method prestate and checking at the end of each method that the receiver's invariant holds [48]—is unsound for multi-object invariants, in the presence of call-backs, and for languages that permit field updates of objects other than the current receiver.

Problem. A *sound* verification technique has to ensure that an object invariant holds in all states in which it is expected to hold. This is achieved by imposing checks (proof obligations) for code that is supposed to establish or preserve an invariant. For instance, most verification techniques impose a check that a constructor establishes the invariant of the fresh object such that later operations on this object may safely assume the invariant. A *modular* verification technique imposes checks that can be verified locally for each class, without knowing its subclasses or clients.

With multi-object invariants, a single field update potentially breaks the invariants of many objects. Consider for instance the third invariant of class `ArrayList` in Fig. 4, which expresses that the elements of the array `elems` are non-null. In general, any array update a`[`j`] = null` potentially breaks this invariant for any `ArrayList` object o where o`.elems`$= a$. A proof obligation for the method m containing this array update would have to quantify over all (relevant) `ArrayList` objects o, which is non-modular because m and class `ArrayList` may have been developed independently, and so `ArrayList` is not known during the verification of m. So the problem is:

How to verify multi-object invariants modularly?

Ownership Solution. Building on the invariant semantics described in Sec. 4.4, multi-object invariants can be verified modularly as follows [37,49,52].

First, we use the ownership topology to restrict the objects an invariant may depend on. An *admissible ownership-based invariant* of an object o may depend

on the fields of o and of all objects (transitively) owned by o[6]. In our example, the third invariant of `ArrayList` is admissible because the array `elems` is owned by the `ArrayList` object. Admissibility can be checked syntactically based on ownership annotations and possibly read effects of pure methods mentioned in invariants (such as method `Length` in class `Map`, see Fig. 7).

Second, we use an encapsulation discipline to restrict the modifications a method may perform. For instance with the owner-as-modifier discipline, when a method m with receiver r updates a field of an object o or an element of an array o then o is either owned by r or it is a peer of r (which subsumes the case $o = r$). In the former case, admissibility guarantees that the update affects at most the invariants of o, r (which owns o), and the (transitive) owners of r. For o and r, we can impose proof obligations that can be checked modularly. For the owners of r, no check is required according to the relevant invariant semantics because those objects are not relevant for r. In the latter case, admissibility guarantees that the update affects at most the invariants of o and the (transitive) owners of o. For o, we can impose a proof obligation that can be checked modularly. The owners of o, which are the owners of r, are not relevant for r.

When m calls an impure method n on a receiver o then again o is either owned by r or it is a peer of r. In the former case, admissibility and the verification of n ensure that all invariants potentially violated by n are checked before n terminates except for the invariants of r and r's (transitive) owners. For r, we can impose a proof obligation on m that can be checked modularly. The owners of r are not relevant for r. In the latter case, admissibility and the verification of n ensure that all invariants potentially violated by n are checked before n terminates except for the invariants of o's (transitive) owners. The owners of o, which are the owners of r, are not relevant for r.

We explained how Spec# enforces a variation of the owner-as-modifier discipline in Sec. 3.2. Using this approach, it is sufficient to check the invariant of an object o at the end of o's constructor as well as at the end of each `expose(`o`)` block. In class `ArrayList` (Fig. 4) the third invariant is checked at the end of the constructor (it holds because `next` is zero and, thus, the quantifier in the invariant ranges over an empty interval[7]) as well as at the end of the `expose` block in method `Add` (it holds because it held at the beginning of the `expose` block and because `e` is non-null according to its type).

Note that the `Add` method of a list l may in general violate the invariant of a `Map` object o using the list. To make o's invariant admissible, o must own l. Hence, o is not relevant for l and need not be checked in method `Add`, which would be non-modular. The check happens instead in the caller of `Add`, for instance, method `Put` (Fig. 7), where o gets exposed before calling `Add`.

The ownership solution to verifying invariants has also been adapted to model fields [38]. In this adaptation, model fields and their representation clauses are encoded as ghost fields with an appropriate invariant, respectively. Instead of

[6] Extra restrictions are required for a sound treatment of inheritance, which we ignore here [37].

[7] Intervals in Spec# are half-open to the right.

checking that the invariant holds, one can update the ghost field automatically such that the invariant is maintained.

Other Solutions. Not all multi-object invariants are protected by encapsulating the state the invariant depends on. *Visibility-based invariants* [49,4] may depend on fields outside the ownership tree as long as the invariant is visible wherever the field can be updated. It is thus possible to generate modular checks at each update. Visibility-based invariants relate for instance the nodes in a doubly-linked list, which do not own each other. *Monotonicity-based invariants* may depend on fields whose value changes monotonically as long as any monotonic change preserves the invariant [45]. For instance, for a counter c that only grows, a monotonicity-based invariant might express $c > 7$. When this invariant has once been established, it will not be violated by any monotonic update of c. A special case of monotonicity-based invariants is invariants over immutable state [42].

Verification methodologies based on dynamic frames [32] and permissions [40,60,64] typically do not support invariants. Consistency criteria are specified explicitly in method pre- and postconditions, with suitable abstraction mechanisms to preserve information hiding [58]. To allow implementations to hide consistency criteria altogether from clients, Region Logic introduced the concept of a dynamic boundary [55]. The *dynamic boundary* of a data structure is a set of objects (or locations) that over-approximates the read effect of the data structure's invariant. Clients of a data structure must not modify objects in the dynamic boundary, which allows the implementation of the data structure to assume the invariant whenever an operation of the data structure is invoked.

5.2 Object Immutability

Immutable data structures simplify many aspects of programming and program verification. For instance:

- Properties of immutable state are trivial to frame because they do not get invalidated by any heap update. This simplifies the specification of read effects and the framing of heap properties (see Sec. 4.2).
- Multi-object invariants may depend on the state of immutable objects without encapsulating them because no method can violate the invariant by modifying an immutable object. This reduces the need to introduce ownership (see Sec. 5.1) and, therefore, allows more sharing.
- Concurrent accesses to immutable objects do not require synchronization because all accesses are read-only. This simplifies the verification of data race freedom (see Sec. 5.3).

Problem. Immutable data structures are typically implemented as immutable classes. An immutable class such as Java's `Integer` encapsulates all its fields, initializes the fields in the constructor, and provides no methods to modify the fields. Therefore, once an object has been initialized, clients have no way of

changing its state. For object structures such as string objects implemented on top of (mutable) character arrays, the class in addition has to ensure that references to mutable sub-objects are not leaked to clients. For instance, Java's `String` class has no method to obtain a reference to the internal character array.

While this idiom is a simple way of implementing immutable data structures, it has several shortcomings for verification. First, immutable object structures may reference mutable objects. For instance, an immutable collection may contain mutable elements. Therefore, it is necessary to delimit the portion of the data structure that is supposed to be immutable. Second, it is not entirely trivial to verify that the implementation actually guarantees immutability, for instance, that references to mutable sub-objects are not leaked and that the object does not escape from its constructor while it is still under initialization (and, thus, being modified) [65]. Third, it is often useful to have immutable instances of a class that otherwise allows mutations, for instance, an immutable instance of `ArrayList`. However, this is not supported by the idiom described above, which requires that a class does not offer any mutating methods. Fourth, some data structures become immutable only after a lengthy initialization phase that goes beyond the execution of the constructor. For instance, an AST might become immutable only after type checking has been completed. Again, this is not supported by immutable classes, which require that an object is fully initialized when the constructor terminates. So the problem is:

How to define and check the immutability of object structures?

Ownership Solution. Ownership provides a natural way to delimit which portion of an object structure is immutable. Ownership-based solutions to immutability [6,25,42,57] define this portion to be the ownership tree rooted in an immutable object. That is, all objects (transitively) owned by an immutable object are also immutable. This definition addresses the first problem mentioned above and supports, in particular, immutable aggregate objects. For instance, when an immutable string object owns its character array, the array also becomes immutable. In our example from Fig. 4 the `elems` array of an immutable `ArrayList` object is also immutable because it is owned by the `ArrayList` object, but the elements stored in the array are not.

The immutability of an ownership tree can be enforced by an encapsulation scheme that prevents modifications of the objects in the tree [6,42,57]. Spec# prevents modifications by assigning immutable objects a special owner, called the *freezer* object[8]. The freezer does not mutate any of the objects it owns and neither can other objects because they would have to expose the freezer first, which is not possible. Since Spec# does not allow an owned object to change owner, an object that is (transitively) owned by the freezer will stay owned by the freezer—and thus immutable—for the rest of its lifetime. Building on an existing encapsulation mechanism allows Spec# to enforce immutability without

[8] The implementation of immutable objects is still experimental [46] and not part of the Spec# release yet.

any additional checks, which addresses the second problem mentioned above. Since ownership and, thus, immutability operate on the object rather than the class level, this solution supports immutable instances of mutable classes, which addresses the third problem mentioned above.

An object is made immutable with a special statement `freeze`, which takes an un-owned object and sets its owner to the freezer. The `freeze` statement can be used anywhere in the code, which allows objects to go through a complex initialization phase before they become immutable. This addresses the fourth problem mentioned above.

```
class Cell {
  int val;

  [return: Frozen] public List SingletonList() {
    ArrayList l = new ArrayList();
    l.Add(this);
    freeze l;
    this.val++;    // okay:  cell is mutable
    l.Add(this);   // verification error: list is immutable
    return l;
  }
}
```

Fig. 8: Method `SingletonList` populates a mutable list and then makes it immutable. The list elements remain mutable because they are not owned by the list. The `Frozen` attribute indicates that the method result is immutable.

The code snippet in Fig. 8 illustrates the ownership solution to immutability. Method `SingletonList` creates and initializes an `ArrayList` object. The subsequent `freeze` statement makes the object immutable. However, since the list does not own its contents, the `Cell` object `this` remains mutable as illustrated by the update of field `val`. Attempting to modify the list after freezing (via the second call to `Add`) results in a verification error because `l` is owned by the freezer. Since the freezer is always valid, `l` is not peer consistent and, thus, does not satisfy the default precondition of `Add` (nor any other precondition that would allow the method to modify the object).

Other Solutions. IGJ [66] uses Java's generic types to check immutability of classes, objects, and references. Objects that are reachable from an immutable object are also immutable, although fields can be explicitly excluded. IGJ provides most of the flexibility of ownership-based solutions, but requires that immutable objects are fully initialized when their constructor terminates, which prevents complex initialization schemes. Later work on OIGJ [67] combines ownership and immutability into a unified system.

5.3 Data Race Freedom

The applications of ownership we have discussed so far are relevant for the verification of both sequential and concurrent programs. However, ownership is also useful to address concurrency-specific problems, in particular, the verification of data race freedom.

Problem. A common way of preventing data races is to synchronize accesses to shared resources through locking. Verification of such programs requires one to prove that no thread accesses a shared resource without holding the lock that protects the resource. This is simple if there is a one-to-one correspondence between resources and locks, for instance, when the locations of an object are protected by the object's lock. However, many programs use more flexible locking patterns such as coarse-grained locking, where one locks protects a whole data structure consisting of many objects. A concurrent (thread-safe) version of our `ArrayList` example might use the lock associated with an `ArrayList` object to protect accesses to this object as well as to the underlying array. To handle flexible locking patterns, the specification must express which locks protect which resources, and the verification must ensure that the required locks are held whenever a thread accesses a shared resource. So the problem is:

How to verify mutual exclusion for accesses to shared resources?

Ownership Solution. A simple way of verifying mutual exclusion for coarse-grained locking is to associate exactly one lock with each ownership tree, which protects the locations of all objects in the tree. Any read or write access to an object in the ownership tree requires acquiring the tree's lock first. This requirement can be checked using an encapsulation discipline that prevents read and write accesses through non-owning references. Such a discipline is stricter than owner-as-modifier because it restricts read and write accesses, but more permissive than owner-as-dominator because it does not restrict the existence of references.

This solution has been implemented in SpecLeuven, an extension of Spec# to concurrency [29] and in VCC [13]. Both systems keep track of the set of objects a thread may safely access. A thread can get read and write access to the root of an ownership tree by locking it. Access to children is obtained by exposing their parent. This discipline is similar to Spec#, but requires exposing the receiver of both read and write accesses. VCC allows programs to implement their own locks (using volatile variables), which then own the objects they protect.

Several type systems use ownership to check data race freedom. For instance Boyapati et al. [7] employ ownership types with the owner-as-dominator discipline to check for data races. Universes for Race Safety [16] check data race freedom using the owner-as-modifier encapsulation discipline. The system associates a lock with each ownership context and therefore supports finer grained locking.

Other Solutions. There are numerous type systems, static analyses, and model checkers to detect data races [50]. In the realm of verification, permission-based systems check for each heap access that the thread has the necessary access permission. Since permissions cannot be forged, these checks ensure mutual exclusion (fractional permissions [9] can be used to allow concurrent reading). Permission-based verification is for instance supported in concurrent separation logic [56], VeriFast [30], and Chalice [40]. Locking strategies can be specified by associating permissions with locks, which requires that a thread must acquire a lock in order to obtain the associated permissions.

6 Conclusion

Program verification was one of the first applications of object ownership and ownership types. Initial work in this area focused on framing [51,49], the verification of object invariants [49], and checking data race freedom [8]. In this paper, we surveyed these and other applications of object ownership to program verification and illustrated how they are supported in the Spec# system.

Since ownership enabled the first sound, modular, and automatic verification techniques for object-oriented programs, a number of alternative approaches have been proposed. Among the most successful are separation logic [60,58] (and other permission-based logics such as Implicit Dynamic Frames [64,40]) and Dynamic Frames [31,32] (and similar logics based on explicit footprints such as Region Logic [1]). These alternatives are generally more flexible than ownership-based verification, especially for non-hierarchical data structures [4] and for data structures whose topology changes over time [61]. However, this flexibility comes at the prize of more annotation overhead, for instance, to specify access permissions or to update ghost state. So for programs that fit the ownership model, ownership-based verification is still a relatively simple and lightweight approach that scales to the verification of intricate concurrent programs as illustrated by VCC [13].

Acknowledgments. We thank the editors for inviting us to contribute to this book, and the reviewers for their helpful feedback.

References

1. A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In J. Vitek, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 5142 of *LNCS*, pages 387–411. Springer-Verlag, 2008.
2. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology (JOT)*, 3(6):27–56, 2004.
3. M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: The Spec# experience. *Communications of the ACM*, 54(6):81–91, June 2011.

4. M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *Mathematics of Program Construction (MPC)*, LNCS. Springer-Verlag, 2004.
5. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *Principles of Programming Languages (POPL)*, pages 259–270. ACM, 2005.
6. C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT, 2004.
7. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 211–230. ACM Press, 2002.
8. C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 56–69. ACM Press, 2001.
9. J. Boyland. Checking interference with fractional permissions. In *Static Analysis Symposium (SAS)*, volume 2694 of *LNCS*, pages 55–72. Springer-Verlag, 2003.
10. D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 292–310. ACM Press, 2002.
11. D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM Press, 1998.
12. D. Clarke and T. Wrigstad. External uniqueness is unique enough. In L. Cardelli, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *LNCS*. Springer-Verlag, 2003.
13. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs)*, pages 23–42. Springer-Verlag, 2009.
14. E. Cohen, M. Moskal, W. Schulte, and S. Tobies. Local verification of global invariants in concurrent programs. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification (CAV)*, volume 6174 of *LNCS*, pages 480–494. Springer-Verlag, 2010.
15. B. Cook, A. Podelski, and A. Rybalchenko. Proving program termination. *Communications of the ACM*, 54:88–98, May 2011.
16. D. Cunningham, S. Drossopoulou, and S. Eisenbach. Universe Types for Race Safety. In *Verification and Analysis of Multi-threaded Java-like Programs (VAMP)*, pages 20–51, 2007.
17. Á. Darvas and K. R. M. Leino. Practical reasoning about invocations and implementations of pure methods. In M. B. Dwyer and A. Lopes, editors, *Fundamental Approaches to Software Engineering (FASE)*, volume 4422 of *LNCS*, pages 336–351. Springer-Verlag, 2007.
18. W. Dietl. *Universe Types: Topology, Encapsulation, Genericity, and Tools*. PhD thesis, Department of Computer Science, ETH Zurich, 2009.
19. W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In E. Ernst, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *LNCS*, pages 28–53. Springer-Verlag, 2007.
20. W. Dietl, S. Drossopoulou, and P. Müller. Separating ownership topology and encapsulation with Generic Universe Types. *Transactions on Programming Languages and Systems (TOPLAS)*, 33:20:1–20:62, 2011.

21. W. Dietl, M. D. Ernst, and P. Müller. Tunable static inference for Generic Universe Types. In *European Conference on Object-Oriented Programming (ECOOP)*, July 2011.
22. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, 2005.
23. S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. A unified framework for verification techniques for object invariants. In J. Vitek, editor, *European Conference on Object-Oriented Programming (ECOOP)*, LNCS. Springer-Verlag, 2008.
24. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI)*, volume 37(5) of *ACM SIGPLAN Notices*, pages 234–245. ACM Press, 2002.
25. C. Haack, E. Poll, J. Schäfer, and A. Schubert. Immutable objects for a Java-like language. In R. De Nicola, editor, *European Symposium on Programming (ESOP)*, volume 4421 of *LNCS*, pages 347–362. Springer-Verlag, 2007.
26. J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. Behavioral interface specification languages. *Computing Surveys*, 2012. To appear.
27. C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.
28. M. Huisman. *Reasoning about Java Programs in higher order logic with PVS and Isabelle*. Ipa dissertation series, 2001-03, University of Nijmegen, Holland, 2001.
29. B. Jacobs, F. Piessens, J. Smans, K. R. M. Leino, and W. Schulte. A programming model for concurrent object-oriented programs. *Transactions on Programming Languages and Systems (TOPLAS)*, 31(1), 2008.
30. B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *Programming Languages and Systems (APLAS 2010)*, pages 304–311. Springer-Verlag, 2010.
31. I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *Formal Methods (FM)*, 2006.
32. I. T. Kassios. The dynamic frames theory. *Formal Aspects of Computing*, 23(3):267–289, 2011.
33. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, and W. Dietl. JML Reference Manual. Available from `http://www.jmlspecs.org/`, June 2008.
34. K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.
35. K. R. M. Leino. Data groups: Specifying the modification of extended state. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, pages 144–153. ACM Press, 1998.
36. K. R. M. Leino. Dafny: an automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 6355 of *LNCS*, pages 348–370. Springer-Verlag, 2010.
37. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.
38. K. R. M. Leino and P. Müller. A verification methodology for model fields. In P. Sestoft, editor, *European Symposium on Programming (ESOP)*, volume 3924 of *LNCS*, pages 115–130. Springer-Verlag, 2006.

39. K. R. M. Leino and P. Müller. Verification of equivalent-results methods. In S. Drossopoulou, editor, *European Symposium on Programming (ESOP)*, volume 4960 of *LNCS*, pages 307–321. Springer-Verlag, 2008.
40. K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In G. Castagna, editor, *European Symposium on Programming (ESOP)*, volume 5502 of *LNCS*, pages 378–393. Springer-Verlag, 2009.
41. K. R. M. Leino and P. Müller. Using the Spec# language, methodology, and tools to write bug-free programs. In P. Müller, editor, *Advanced Lectures on Software Engineering—LASER Summer School 2007/2008*, volume 6029 of *LNCS*, pages 91–139. Springer-Verlag, 2010.
42. K. R. M. Leino, P. Müller, and A. Wallenburg. Flexible immutability with frozen objects. In *Verified Software: Theories, Tools, and Experiments (VSTTE)*, volume 5295 of *LNCS*, pages 192–208. Springer-Verlag, 2008.
43. K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *Transactions on Programming Languages and Systems (TOPLAS)*, 24(5):491–553, 2002.
44. K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Programming Language Design and Implementation (PLDI)*, volume 37(5) of *ACM SIGPLAN Notices*, pages 246–257. ACM Press, 2002.
45. K. R. M. Leino and W. Schulte. Using history invariants to verify observers. In R. De Nicola, editor, *Programming Languages and Systems (ESOP)*, volume 4421 of *LNCS*, pages 80–94. Springer-Verlag, 2007.
46. F. Leu. Implementation of frozen objects into Spec#. Master's thesis, ETH Zurich, 2009. Available from `http://www.pm.inf.ethz.ch/education/theses/student_docs/Florian_Leu/florian_leu_MA_report`.
47. Y. Lu, J. Potter, and J. Xue. Validity invariants and effects. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *LNCS*, pages 202–226. Springer-Verlag, 2007.
48. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.
49. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.
50. P. Müller. Formal methods-based tools for race, deadlock and other errors. In D. Padua, editor, *Encyclopedia of Parallel Computing*, pages 704–710. Springer-Verlag, 2011.
51. P. Müller and A. Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*. Cambridge University Press, 2000.
52. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.
53. P. Müller and A. Rudich. Ownership transfer in Universe Types. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 461–478. ACM Press, 2007.
54. D. A. Naumann. Observational purity and encapsulation. *Theor. Comput. Sci.*, 376(3):205–224, 2007.
55. D. A. Naumann and A. Banerjee. Dynamic boundaries: Information hiding by second order framing with first order assertions. In A. D. Gordon, editor, *Programming Languages and Systems (ESOP)*, volume 6012 of *LNCS*, pages 2–22. Springer-Verlag, 2010.
56. P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375:271–307, 2007.

57. J. Östlund, T. Wrigstad, D. Clarke, and B. Åkerblom. Ownership, uniqueness, and immutability. In R. F. Paige and B. Meyer, editors, *Objects, Components, Models and Patterns (TOOLS)*, volume 11 of *Lecture Notes in Business Information Processing*, pages 178–197. Springer-Verlag, 2008.
58. M. Parkinson and G. Bierman. Separation logic and abstraction. In J. Palsberg and M. Abadi, editors, *Principles of Programming Languages (POPL)*, pages 247–258. ACM Press, January 2005.
59. A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, Jan. 1997.
60. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*. IEEE Computer Society Press, 2002.
61. A. Rudich. *Automatic Verification of Heap Structures with Stereotypes*. PhD thesis, ETH Zurich, 2011.
62. A. Rudich, Á. Darvas, and P. Müller. Checking well-formedness of pure-method specifications. In J. Cuellar and T. Maibaum, editors, *Formal Methods (FM)*, volume 5014 of *LNCS*, pages 68–83. Springer-Verlag, 2008.
63. A. Salcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In *VMCAI*, volume 3385 of *LNCS*, pages 199–215. Springer-Verlag, 2005.
64. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *European Conference on Object-Oriented Programming (ECOOP)*, LNCS, pages 148–172. Springer-Verlag, 2009.
65. A. J. Summers and P. Müller. Freedom before commitment—a lightweight type system for object initialisation. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1013–1032. ACM, 2011.
66. Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kieżun, and M. D. Ernst. Object and reference immutability using Java generics. In *European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 75–84. ACM Press, 2007.
67. Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst. Ownership and immutability in generic Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 598–617, 2010.