Object Invariants in Dynamic Contexts

K. Rustan M. Leino [1] and Peter Müller [2]

[1] Microsoft Research, Redmond, WA, USA, leino@microsoft.com
[2] ETH Zurich, Switzerland, peter.mueller@inf.ethz.ch

Abstract. Object invariants describe the consistency of object-oriented data structures and are central to reasoning about the correctness of object-oriented software. Yet, reasoning about object invariants in the presence of object references, methods, and subclassing is difficult. This paper describes a methodology for specifying and verifying object-oriented programs, using object invariants to specify the consistency of data and using ownership to organize objects into contexts. The novelty is that contexts can be dynamic: there is no bound on the number of objects in a context and objects can be transferred between contexts. The invariant of an object is allowed to depend on the fields of the object, on the fields of all objects in transitively-owned contexts, and on fields of objects reachable via given sequences of fields. With these invariants, one can describe a large variety of properties, including properties of cyclic data structures. Object invariants can be declared in or near the classes whose fields they depend on, not necessarily in the class of an owning object. The methodology is designed to allow modular reasoning, even in the presence of subclasses, and is proved sound.

1 Introduction

The design and correctness of computer programs rely on *invariants*, consistency conditions on the program's data that are to be maintained throughout the execution of the program. In object-oriented programs, which permit a flexible and extensible organization of data and control, the invariants are dominated by *object invariants*, which relate the data fields of objects. Fig. 1 illustrates the use of an object invariant in a simple class, whose correctness relies on the object invariant to hold on entry to the method m. In this paper, we consider the specification and verification of object-oriented programs with object invariants.

The dynamic nature of object-oriented programs makes it difficult to construct a systematic technique for reasoning modularly about object invariants. Such a technique would allow one to verify the correctness of a class, or a set of classes, independently of possible subclasses and other parts of the program. A central problem is that an object invariant may temporarily be violated during the update of an object's data, and any method call performed by the update code during this time may potentially cause control to re-enter the object's public interface where the object invariant is expected to hold. For example, in Fig. 1, $a = b$ may hold at the time P is called, a temporary violation of the object invariant which results in a division-by-zero error if P calls back into m. Recently, some progress has been made in tackling this problem (*e.g.*, [2, 30]), but more progress is required to handle more programs.

To make reasoning easier, various restrictions on the dynamism of programs have been considered, including *alias confinement* techniques which impose restrictions on

```
class T {
  int a, b ;
  invariant 0 ⩽ a < b ;
  public T( ) { a := 0 ; b := 3 ; }
  public void m(...) {
    int k := 100/(b − a) ;
    a := a + 3 ; P(...) ; b := (k + 4) ∗ b ;
  }
}
```

Fig. 1. A simple class illustrating the use of an object invariant. The invariant is to be established by the constructor and maintained by the method m. Upon entry to m, the invariant implies the absence of a division-by-zero error. The fact that the invariant may not hold at the time m calls procedure P is a central problem in reasoning: if P calls back into m, then m may erroneously divide by 0.

a program's object references. A promising alias confinement approach is based on *ownership* (*e.g.*, [9, 4, 31, 5]), where an object is considered to *own* the objects that form parts of its data representation, its *constituent objects*. The methodology we present in this paper uses ownership to organize objects into a hierarchy of *contexts*, where the objects in each context have a common owner. In our methodology, an owner is a pair consisting of an object reference and a class name. Other than ownership, we do not restrict object references; an object may have non-owning references to other objects. Our methodology allows *ownership transfer*, whereby the owner of an object is changed during program execution, that is, whereby an object changes contexts.

An important consideration in a methodology for object invariants is what object fields an invariant is allowed to depend on. We allow object invariants to depend on three sorts of object fields. First, the invariant declared for an object X in a class T is allowed to depend on the fields of X declared in any superclass of T (here and throughout, we use "superclass" and "subclass", unless further qualified, to include the class itself). Second, the invariant is allowed to depend on the fields of any object transitively owned by $[X, S]$ for any superclass S of T. We allow an invariant to quantify over these owned objects, which means that the invariant can depend on the fields of an unbounded number of objects. Moreover, because we allow such quantifications, X's invariant can depend on the fields of owned objects even for owned objects that are not reachable from X in the heap. Third, our methodology allows the invariant of X to depend on the fields of any specified object $X.f_1.\cdots.f_n$ ($n \geqslant 1$)—that is, an object reachable from X by a fixed sequence of field dereferences—provided certain visibility requirements are met. Altogether, this set of permissible object invariants is larger than those allowed by previous work. The invariants can, for example, describe properties of cyclic data structures and they can be declared in or near the classes whose fields they depend on.

In the next section, we summarize the previous work that forms a basis for our approach. In Section 3, we describe our ownership model and show how object invariants are checked. Section 4 describes visibility-based invariants and shows how these can be declared and checked within local portions of a context. In Section 5, we give the technical encoding of our methodology and prove a soundness theorem. Section 6 discusses usability aspects of our methodology, and the paper ends with related work, conclusions, and future work.

$\mathbf{class}\ C\ \mathbf{extends}\ B\ \{\ \mathbf{int}\ w\ ;\ \mathbf{invariant}\ w < 100\ ;\ \ldots\ \}$
$\mathbf{class}\ B\ \mathbf{extends}\ A\ \{\ \mathbf{int}\ z\ ;\ \mathbf{invariant}\ y < z\ ;\ \ldots\ \}$
$\mathbf{class}\ A\ \mathbf{extends}\ \mathbf{object}\ \{\ \mathbf{int}\ x, y\ ;\ \mathbf{invariant}\ x < y\ ;\ \ldots\ \}$

C:	$w = 43$
B:	$z = 6$
A:	$x = 5 \;\; y = 7$
$\mathbf{object}$:	$inv = A \;\; \ldots$

Fig. 2. Given the declarations of classes A, B, and C, which include field and invariant declarations, the object of allocated type C depicted to the right is in a possible state. In particular, since $inv = A$, the invariant declared in class A is known to hold, whereas the other invariants may or may not hold.

2 Approach

Our approach combines two previous techniques, summarized in this section, to produce a methodology for specifying and verifying object invariants that is more flexible than any previous sound, modular technique. In this paper, we consider a Java-like object-oriented language, but omit treatment of static fields (global variables) and concurrency.

2.1 Explicit representation of which parts of object invariants are known to hold

One of the previous methodologies on which we build ours is that of Barnett *et al.* [2]. In that methodology, whether or not an object invariant is known to hold is explicitly represented in the program's state and an ownership model is enforced through a set of constrained assignments to two special object fields (inv and $committed$, described next). Here, we summarize this previous methodology; for a full description of its rationale, see the other paper [2].

Declarations of object invariants can appear in every class. The invariants that pertain to an object o are those declared in the classes between $\mathbf{object}$—the root of the single-inheritance hierarchy—and $\mathbf{type}(o)$—the allocated type of o. Each object o has a special field inv, whose value names a class in the range from $\mathbf{object}$ to $\mathbf{type}(o)$, and which represents the most refined subclass whose invariant can be relied upon for this object. More precisely, for any object o and class T and in any execution state of the program, if $o.inv$ is a subclass of T, which we denote by $o.inv \leqslant T$, then all object invariants declared in class T are known to hold for o. The object invariants declared in other classes may or may not hold for o, so they cannot be relied upon. For example, Fig. 2 shows a possible state of an object of allocated type C. If $o.inv$ equals $\mathbf{type}(o)$, then we say the object is *consistent*. Thus, all invariants hold of consistent objects. For example, the object depicted in Fig. 2 is not consistent.

As part of the ownership model of the methodology, each object also has a special boolean field $committed$, representing whether or not the object is owned. Moreover, the program's field declarations can be tagged with the modifier $\mathbf{rep}$. An object o is the owning object of p, that is, object p is owned by o, if and only if p is committed and there is a rep field f declared in a class T such that $o.inv \leqslant T$ and $o.f = p$. Only consistent objects can be committed (that is, $p.committed$ implies $p.inv = \mathbf{type}(p)$), and a committed object has exactly one owning object.

The special fields inv and $committed$ can be mentioned in routine specifications, but cannot be mentioned in object invariants and cannot directly be read or updated by the code of the program. Instead, two program statements, **pack** and **unpack**, are introduced for the purpose of changing the values of the two special fields. For any class T with immediate superclass S and any object expression o of type T, these statements are defined as

$$
\begin{array}{l}
\textbf{pack } o \textbf{ as } T \ \equiv \\
\quad \textbf{assert } o \neq \textbf{null} \wedge o.inv = S \ ; \\
\quad \textbf{assert } Inv_T(o) \ ; \\
\quad \textbf{foreach } p \in Constit_T(o) \ \{ \ \textbf{assert } p.inv = \textbf{type}(p) \wedge \neg p.committed \ \} \\
\quad \textbf{foreach } p \in Constit_T(o) \ \{ \ p.committed := true \ \} \\
\quad o.inv := T \\
\textbf{unpack } o \textbf{ from } T \ \equiv \\
\quad \textbf{assert } o \neq \textbf{null} \wedge o.inv = T \wedge \neg o.committed \ ; \\
\quad o.inv := S \ ; \\
\quad \textbf{foreach } p \in Constit_T(o) \ \{ \ p.committed := false \ \}
\end{array}
$$

where the **assert** statements give the conditions under which the **pack** and **unpack** statements are legal, the assignment statements describe the effects of the **pack** and **unpack** statements, $Inv_T(X)$ is the condition that says an object X satisfies the object invariant declared in class T, $Constit_T(X)$ is the set of expressions $X.f$ for all rep fields f declared in T, and the **foreach** statements, like all quantifications over object references in this paper, range over allocated non-null objects. In words, the **pack** statement checks that o's T-invariant holds, that the objects referenced by o's rep fields in T are consistent and uncommitted, marks o's T-constituent object as being committed, and records the fact that now o's T-invariant is known to hold. The **unpack** statement records the fact that o is now in a state where its T-invariant may be violated and decommits o's T-constituent objects.

The methodology ensures that the following two conditions are *program invariants*—that is, they hold in every reachable program state—for every class T:

$$
\begin{array}{l}
(\forall\, o \bullet \ o.inv \leqslant T \ \Rightarrow \ Inv_T(o) \wedge (\forall\, p \in Constit_T(o) \bullet \ p.committed \ )) \\
(\forall\, p \bullet \ p.committed \ \Rightarrow \ p.inv = \textbf{type}(p) \ )
\end{array}
$$

Three more things are needed to guarantee that these conditions are program invariants. First, the methodology prescribes the **object** constructor to have the postcondition $inv = \textbf{object} \wedge \neg committed$. Second, for any field f declared in a class T, a field update statement $o.f := e$ is legal only if $T < o.inv$. If the class T is understood from context, then we may refer to the condition "$T < o.inv$" as "*o is sufficiently unpacked*". That is, a precondition for updating $o.f$ is that o is sufficiently unpacked. Third, an invariant is admissible only if all of the field-access expressions it mentions have the form $\textbf{this}.g_1.\cdots.g_n.f$, where **this** denotes the object whose invariant is being described, $n \geqslant 0$, and the fields $g_1, \ldots, g_n$ are all rep fields. (And, of course, all statements and expressions are subject to ordinary typechecking.) Like in Java, the prefix "**this**." can be omitted.

Note that this methodology permits an object invariant to depend on fields declared in superclasses and on fields of transitively-owned objects. However, because an object

is an owned object only if it is referenced by a rep field, there is a static limit on the number of owned objects an owning object can have. In particular, the number of objects with owner $[X, T]$ is bounded by the number of rep fields declared in class T. Our methodology in this paper removes this limitation.

Note also that the encoding of the methodology records only which objects are committed, not the owning objects to which they are committed. In this paper, we extend the encoding also to record the owner.

Finally, note that this methodology allows ownership transfer. For example, the following code sequence, where f is a rep field declared in a class T and r and o are distinct non-null objects of type T, transfers from o to r the ownership of the object initially referenced by $o.f$:

$$\begin{array}{l}\textbf{unpack } o \textbf{ from } T \text{ ; } \textbf{unpack } r \textbf{ from } T \text{ ;}\\ r.f := o.f \text{ ; } \textbf{pack } r \textbf{ as } T \text{ ;}\\ o.f := \textbf{null} \text{ ; } \textbf{pack } o \textbf{ as } T\end{array}$$

2.2 Universe types

The other of the previous methodologies on which we build ours is that in Müller's thesis [30, 32, 31]. In that methodology, an ownership type system organizes objects into contexts, called *universes*, and object invariants are specified as the representation of special boolean *abstract fields*, which are often-underspecified functions of actual object fields. The state of a universe is *encapsulated*, that is, the fields of the universe's objects can be modified only when control is in a method applied to an object within the universe.

A universe can have several owner objects, all of which belong to the enclosing universe. The invariant of an object o is allowed to depend on the fields of all objects in o's universe and in all nested universes. In particular, the invariant may depend on fields of objects that are not transitively owned by o. Such invariants are enabled by imposing a *visibility requirement* [24, 30] that essentially requires that an invariant be visible in every method that might violate the invariant of an object in the universe in which the method executes. This requirement allows one to use the declaration of the invariant to show that invariants are preserved.

In this paper, we too allow invariants to depend on non-owned fields (unlike the methodology by Barnett *et al.*), provided an appropriate visibility requirement is met. Because we don't have (or need) the encapsulation provided by universes, we force certain declarations to be mutually visible by making sure they refer to each other (using a $\mathbf{dependent}$ clause, as we shall see later).

A limitation with this previous methodology is that objects cannot be partially unpacked, to use the parlance of the previous subsection. That is, an object is either in a state where all its invariants (which may be declared in several subclasses) are known to hold or in a state where all of these invariants are allowed to be violated, but never anything in between. Therefore, it is not possible to reason separately about the object invariants declared in different subclasses. In this paper, we overcome this limitation by following the methodology by Barnett *et al.*

Another limitation with this previous methodology is that call-backs from a nested universe to an enclosing universe are disallowed, except for calling so-called *pure* methods, which are not allowed to rely on the object invariant and are not allowed to modify the program's state. This limitation comes about because a nested universe has no way of knowing whether or not the invariants in an enclosing universe hold. In this paper, we overcome this limitation by explicitly representing when an object's invariants hold.

A third limitation with this previous methodology is that the type system that tracks ownership does not allow ownership transfer. As we shall see in this paper, visibility-based invariants also complicate the situation with ownership transfer, but we are able to overcome the limitation.

3 Ownership

In this section, we explain the basics of our methodology.

Following Barnett *et al.*, our methodology uses an explicit representation, namely the special fields inv and $committed$, of when object invariants are known to hold. In addition, we explicitly encode the ownership relation itself, using a special field $owner$. The value of $owner$ is a pair $[\mathbf{object}\ obj, \mathbf{type}\ typ]$, representing by obj the owning object and by typ the class of the owning object that induces the ownership. If $p.owner = [o, T]$ for a non-null o, then committing p means committing it to o at class T. More precisely, if $p.owner = [o, T]$, then $p.committed$ if and only if $o.inv \leqslant T$. We also allow $p.owner.obj$ to be $\mathbf{null}$, in which case p has no owning object and the value of $p.owner.typ$ is not used.

The $owner$ field can be mentioned in routine specifications and object invariants, but it cannot directly be read or updated by the code of the program. The $owner$ field is initialized by the $\mathbf{object}$ constructor, which takes an owner as a parameter:

$$
\begin{array}{l}
\mathbf{class}\ \mathbf{object}\ \{ \\
\quad \mathbf{object}(\mathbf{pair}[\mathbf{object}, \mathbf{type}]\ ow) \\
\qquad \mathbf{requires}\ ow.obj \neq \mathbf{null} \Rightarrow \mathbf{type}(ow.obj) \leqslant ow.typ < ow.obj.inv\ ; \\
\qquad \mathbf{ensures}\ owner = ow \wedge inv = \mathbf{object} \wedge \neg committed\ ;
\end{array}
$$

We use $\mathbf{requires}$ clauses to declare preconditions and $\mathbf{ensures}$ clauses to declare postconditions. Note that the owning object is required to be sufficiently unpacked, that is, the invariants declared in the owning class must not be assumed to hold for the owning object. The $owner$ field can be updated only by the ownership transfer statement, described below.

Since our methodology includes the field $owner$, we replace the definitions of $\mathbf{pack}$ and $\mathbf{unpack}$ from Section 2.1 as follows. In our methodology, the statement $\mathbf{pack}\ o\ \mathbf{as}\ T$ commits every object that claims $[o, T]$ as its owner, that is, every object whose $owner$ field equals $[o, T]$. Formally, for any class T with immediate superclass S and any object expression o of type T, our methodology defines the $\mathbf{pack}$ and $\mathbf{unpack}$ statements as follows (note that we don't need the somewhat awkward $Constit_T$ construction from Section 2.1):

$$
\begin{array}{l}
\textbf{pack } o \textbf{ as } T \ \equiv \\
\quad \textbf{assert } o \neq \textbf{null} \wedge o.inv = S \ ; \\
\quad \textbf{assert } Inv_T(o) \ ; \\
\quad \textbf{assert } (\forall \textbf{ object } p \mid p.owner = [o, T] \bullet p.inv = \textbf{type}(p) \ ) \ ; \\
\quad \textbf{foreach object } p \mid p.owner = [o, T] \ \{ \ p.committed := true \ \} \\
\quad o.inv := T \\
\textbf{unpack } o \textbf{ from } T \ \equiv \\
\quad \textbf{assert } o \neq \textbf{null} \wedge o.inv = T \wedge \neg o.committed \ ; \\
\quad o.inv := S \ ; \\
\quad \textbf{foreach object } p \mid p.owner = [o, T] \ \{ \ p.committed := false \ \}
\end{array}
$$

The owner of an object o can be changed to $[p, T]$ by the ownership transfer statement, defined as follows:

$$
\begin{array}{l}
\textbf{transfer } o \textbf{ to } [p, T] \ \equiv \\
\quad \textbf{assert } o \neq \textbf{null} \wedge o.inv = \textbf{object} \ ; \\
\quad \textbf{assert } o.owner.obj \neq \textbf{null} \Rightarrow o.owner.typ < o.owner.obj.inv \ ; \\
\quad \textbf{assert } p \neq \textbf{null} \Rightarrow T < p.inv \ ; \\
\quad o.owner := [p, T]
\end{array}
$$

Note that both the old and new owning objects are required to be sufficiently unpacked. We define the transfer statement to typecheck only if T is a superclass of the type of the expression p. Moreover, we require that the type of the expression o be a class tagged with the modifier **transferable**. For now, one can think of this modifier as giving programmers more control, in a way similar to Java's *Cloneable* and *Serializable* interfaces; later, we shall find a more prominent use of the **transferable** modifier. The **transferable** modifier is inherited, that is, subclasses of a transferable class are transferable as well. The predefined class **object** is not transferable.

The invariant of an object o in a class T is allowed to depend on the fields of o declared in any superclass of T and on the fields of any objects transitively owned by $[o, S]$ for any superclass S of T.

We allow fields to be tagged with the **rep** modifier. The declaration of a rep field f in a class T gives rise to an implicit object invariant in class T:

$$\textbf{invariant this}.f \neq \textbf{null} \Rightarrow \textbf{this}.f.owner = [\textbf{this}, T] \ ;$$

Later in the paper, we will also use the **rep** modifiers to formulate a syntactic restriction for policing admissible invariants.

3.1 Example: Invariants of a doubly-linked list

Let us give an example that exhibits the expressiveness of the invariants allowed by our methodology. Fig. 3 shows a class *List* that represents lists of integers. Each list is represented by a doubly-linked set of *Node* objects. To simplify the implementation, the *head* field of a list references a dummy node, where $head.next$ is the first actual element of the list and $head.prev$ is **null**.

All of the *Node* objects that are part of the representation of a *List* object are owned by the *List* object. This is specified by the **rep** modifier on the field *head* and

```
class List {
  rep Node head ;
  invariant head ≠ null ∧ head.prev = null ;
  invariant ( ∀ Node n | n.owner = [this, List] • wf(n) ) ;
  void Insert(int x)
    requires ¬committed ∧ inv = List ;
    ensures ( ∀ object x | ¬old(x.alloc) • x.inv = type(x) ) ;
    {
      unpack this from List ;
      head.Insert(x) ;
      pack this as List ;
    }
  /* Constructors and other methods are omitted */
}

final class Node {
  int val ;
  Node prev ;
  Node next ;
  invariant next ≠ this ∧ prev ≠ this ;
  void Insert(int x)
    requires ¬committed ;
    requires ( ∀ Node n | n.owner = this.owner • wf(n) ∧ n.inv = Node ) ;
    modifies {X.prev, X.next | X.owner = this.owner} ;
    ensures ( ∀ Node n | n.owner = this.owner • wf(n) ∧ n.inv = Node ) ;
    ensures ( ∀ Node n | n.owner = this.owner •
                  old(n.alloc) ∧ old(n.prev) = null ⇒ n.prev = null ) ;
    ensures ( ∀ object x | ¬old(x.alloc) • x.inv = type(x) ) ;
    {...}
  Node(int x, Node n, pair[object, type] ow)
    requires n ≠ null ⇒
      ¬n.committed ∧ n.inv = Node ∧ n.next ≠ n ∧ n.owner = ow ;
    requires ow.typ < ow.obj.inv ;
    modifies n.prev ;
    ensures prev = null ∧ next = n ∧ ¬committed ∧ owner = ow ∧ wf(this) ;
    ensures n ≠ null ⇒ n.prev = this ∧ wf(n) ;
    ensures ( ∀ object x | ¬old(x.alloc) • x.inv = type(x) ) ;
    {...}
}
```

Fig. 3. An example of ownership-based invariants. *List* objects are represented by doubly-linked *Node* objects. The object invariant in *List* specifies which *Node* objects the list owns and specifies properties about those *Node* objects. Class *Node* illustrates that, without visibility-based invariants, *Node* methods need very strong specifications to enable one to verify that the calling *List* methods preserve the *List* invariant.

by (part of) the quantification in the $List$ invariant, where we have used the following abbreviation:

$$wf(n) \equiv (n.next \neq \mathbf{null} \Rightarrow n.next.owner = n.owner \wedge n.next.prev = n) \wedge (n.prev \neq \mathbf{null} \Rightarrow n.prev.owner = n.owner \wedge n.prev.next = n)$$

The invariant also specifies that the $next$ and $prev$ fields of adjacent nodes correspond.

The implementation of $List.Insert$ simply unpacks the list and then calls $Node.Insert$ on the dummy node. When $List.Insert$ then re-packs the list, it needs to know that the $List$ invariant holds, a fact that follows from the rather complicated postcondition of $Node.Insert$.

The postcondition of $List.Insert$ says that all objects allocated from the time the method is entered are, upon exit, consistent. We use the special field $alloc$ to indicate whether or not an object has been allocated. This is useful in postconditions, where $\mathbf{old}(E)$ gives the value of expression E in the method's pre-state. Recall that quantifications over object references range over allocated non-null objects.

In addition to **requires** and **ensures** clauses, a routine specification can include **modifies** clauses. The set of given **modifies** clauses contributes to the postcondition of the routine, constraining what can be modified. We follow Barnett *et al.* to say that a routine is allowed to modify those object fields explicitly indicated by a **modifies** clause, the fields of newly allocated objects, and the fields of objects that are committed in the routine's pre-state (see [2] for the rationale behind this design). That's why, for example, $List.Insert$ has an empty set of **modifies** clauses. The **modifies** clause of $Node.Insert$ uses a set comprehension where X is the bound variable.

Summarizing the example, our methodology allows an object's invariant to depend on the fields of all objects that it owns. Since a list's context is dynamic, this example is not handled by the encoding of Barnett *et al.* where owners are not explicated. However, in what we've presented so far, a problem is that the specifications of routines that are executed while the owning object is unpacked can be rather complicated. Another problem is that there are subtle issues in formally determining whether or not the invariant in $List$ is admissible (to define which $Node$ objects are owned by a list, the $List$ invariant has to depend on the $owner$ field of the object it says it owns). The visibility-based invariants that we introduce in the next section overcome the first of these problems by allowing specifications to be stated more locally. To overcome the second problem, we later introduce a modifier **peer**, which, analogously to the modifier **rep**, gives an indirect way of specifying the ownership part of $List$'s invariant (in fact, our syntactic rules for admissible invariants will then disallow the explicit definition of ownership in the Fig. 3 $List$ invariant).

3.2 Example: Ownership transfer

Fig. 4 shows a method that transfers the ownership of a $Possession$ object. Type checking requires the $Possession$ class to be declared as **transferable**. To be able to unpack $possn$ by a single unpack operation, we require it to be an instance of class $Possession$. To allow arbitrary subclass objects, one would have to unpack $possn$ by a

```
transferable class Possession {...}
class Person {
  rep Possession possn ;
  void donateTo(Person p)
    requires ¬committed ∧ inv = Person ;
    requires possn ≠ null ∧  ∧ type(possn) = Possession ;
    requires p ≠ null ∧ p ≠ this ∧ ¬p.committed ∧ p.inv = Person ;
    modifies possn, p.possn ;
    {
      unpack this from Person ;  unpack p from Person ;
      unpack possn from Possession ;
      transfer possn to [p, Person] ;
      pack possn as Possession ;
      p.possn := possn ;  pack p as Person ;
      possn := null ;  pack this as Person ;
    }
  ...
}
```

Fig. 4. The $donateTo$ method shows that objects can be transferred from one owner to another, provided the old and new owners are sufficiently unpacked. The class $Possession$ is tagged with the modifier **transferable**, indicating that its objects may undergo ownership transfers.

dynamically-bound method that is overridden for each subclass of $Possession$ and unpacks the object step by step. (For more information about such issues with subclasses and specification support thereof, see [2].)

The transfer statement requires both the old owner and the new owner to be sufficiently unpacked. The $possn$ field of **this** is set to **null** to re-establish the implicit invariant (induced by the **rep** modifier on $possn$) that the object referenced by **this**.$possn$ is owned by **this**, which is not the case immediately following the transfer.

Note that our methodology deems the code at the end of Section 2.1 to be incorrect, because without using a **transfer** statement, $r.f.owner$ would still be $[o, T]$, not the required $[r, T]$, before the packing of r.

4 Visibility-based invariants

In this section, we generalize the ownership-based methodology of the previous section to allow object invariants to express properties of the state of objects in arbitrary contexts, including peer objects—objects with the same owner. Soundness is achieved by imposing a syntactic visibility requirement as well as stronger proof obligations for field updates and ownership transfers.

4.1 Limitations of ownership-based invariants

The ownership-based invariants of the previous section allow object invariants to express properties of owned objects, such as the nodes in the doubly-linked list example. Such specifications are typical if the class of the owned objects (*e.g.,* $Node$) comes

from a library and does not provide invariants that are strong enough for the context in which the class is reused. Additional invariants can then only be specified in the next abstraction layer, that is, in the class of the owner object ($List$).

However, insisting that all invariants about the owned objects be expressed in the owner has several shortcomings. We illustrate them by comparing the ownership-based solution for the doubly-linked list (Fig. 3) with an alternative implementation where invariants are specified locally in class $Node$ (Fig. 5).

The **peer** modifier in the declarations of the $next$ and $prev$ fields expresses that the $Node$ objects X, $X.next$, and $X.prev$ are *peers*, that is, they have the the same owner. Analogously to **rep** modifiers, **peer** declarations lead to implicit invariants like the following for class $Node$:

$$\textbf{invariant } \mathbf{this}.next \neq \mathbf{null} \Rightarrow \mathbf{this}.owner = \mathbf{this}.next.owner\ ;$$
$$\textbf{invariant } \mathbf{this}.prev \neq \mathbf{null} \Rightarrow \mathbf{this}.owner = \mathbf{this}.prev.owner\ ;$$

Such invariants cannot be handled by the methodology described so far (note that the invariant depends on fields of $\mathbf{this}.next$ but $\mathbf{this}.next$ is not owned by $[\mathbf{this}, Node]$), but are handled by the generalization presented in this section.

With ownership-based invariants, verification of $List$ methods involves reasoning about properties of the underlying node structure. That is, the modifications of $Node$ objects are not reasoned about locally in the $Node$ class. This lack of locality blurs the interface between different layers of abstraction, which leads to two problems:

1. *Complicated method specifications:* Method specifications of class $Node$ must be strong enough to enable one to show that the calling $List$ methods preserve the invariant. As illustrated by method $Node.Insert$ in Fig. 3, one essentially has to repeat the well-formedness property of $Node$ objects in every pre- and postcondition, which is not necessary in the alternative implementation.
2. *Bulky reasoning:* In order to verify that $List$ methods preserve the ownership-based invariant, one has to consider *all* nodes owned by the list. With the local invariant of the alternative $Node$ implementation, modification of one node can affect only the invariants of its predecessor and successor nodes. That is, showing that invariants are preserved does not involve universal quantifications but only properties of directly referenced objects, which can simplify reasoning.

In the rest of this section, we explain how we extend the methodology of the previous section to support invariants like in the alternative $Node$ implementation.

4.2 Example: Invariants over peer objects

A field update may cause an invariant to be violated. In the presence of just ownership-based invariants, we ensure that no invariant is violated at an inappropriate time by making sure the updated object is sufficiently unpacked, which implies that all transitive owner objects are unpacked. However, to allow invariants to refer to objects in arbitrary contexts, not just transitively owned contexts, we need additional requirements to ensure that *all* objects whose invariants might be affected by the modification are sufficiently unpacked, not only the updated object and its owner objects.

```
class List {
  rep Node head ;
  invariant head ≠ null ∧ head.prev = null ;
  ...
}

final class Node {
  int val ;
  peer Node prev dependent Node ;
  peer Node next dependent Node ;
  invariant (next ≠ null ⇒ next.prev = this) ∧ next ≠ this ;
  invariant (prev ≠ null ⇒ prev.next = this) ∧ prev ≠ this ;
  void Insert(int x)
    requires ¬committed ;
    requires (∀ Node n | n.owner = this.owner • n.inv = Node ) ;
    modifies {X.prev, X.next | X.owner = this.owner} ;
    ensures (∀ Node n | n.owner = this.owner •
      old(n.alloc) ∧ old(n.prev) = null ⇒ n.prev = null ) ;
    ensures (∀ object x | ¬old(x.alloc) • x.inv = type(x) ) ;
    {
      if (next ≠ null ∧ ...)   { next.Insert(x) ; }
      else {
          unpack this from Node ;
          next := new Node(x, next, this.owner) ;
          unpack next from Node ; next.prev := this ; pack next as Node ;
          pack this as Node ;
        }
    }
  Node(int x, Node n, pair[object, type] ow)
    requires n ≠ null ⇒
      ¬n.committed ∧ n.inv = Node ∧ n.prev.inv = object ∧ n.owner = ow ;
    requires ow.typ < ow.obj.inv ;
    modifies n.prev ;
    ensures prev = null ∧ next = n ∧ (n ≠ null ⇒ n.prev = this) ;
    ensures ¬committed ∧ inv = Node ∧ owner = ow ;
    ensures (∀ object x | ¬old(x.alloc) • x.inv = type(x) ) ;
    {
      super(ow) ; val := x ; prev := null ; next := n ;
      if (n ≠ null)
        { unpack n from Node ; n.prev := this ; pack n as Node ; }
      pack this as Node ;
    }
}
```

Fig. 5. The alternative specification of class *List* has a simpler invariant, since the well-formedness of the list nodes is specified locally in class *Node*. The alternative implementation of class *Node* uses **peer** declarations to express that predecessor and successor nodes belong to the same owner as **this**. The **dependent** declarations are necessary to allow invariants of peer objects to depend on fields of **this**.

To illustrate these requirements, we revisit the $Person$ example and add a $spouse$ field as well as a $marry$ method (see Fig. 6). The invariant states that the spouse of a $Person$ object's spouse is the object itself. This invariant is very similar to the well-formedness of nodes, but is easier to verify.

```
class Person {
  rep Possession possn ;
  peer Person spouse dependent Person ;
  owner-dependent Person ;
  invariant this.spouse ≠ null ⇒ this.spouse.spouse = this ;
  ...
  void marry(Person p)
    requires p ≠ this ∧ ¬committed ∧ inv = Person ∧ spouse = null ;
    requires p ≠ null ∧ ¬p.committed ∧ p.inv = Person ∧ p.spouse = null ;
    modifies spouse, p.spouse ;
    {
      unpack this from Person ;  unpack p from Person ;
      this.spouse := p ;  p.spouse := this ;
      pack p as Person ;  pack this as Person ;
    }
}
```

Fig. 6. Method $Person.marry$ requires that neither person already has a spouse, which guarantees that the assignments to the $spouse$ fields do not break invariants of other $Person$ objects.

Field updates $Person$ contains an invariant that refers to $\mathbf{this}.spouse.spouse$, which is a field of a peer object of $\mathbf{this}$. The assignment $\mathbf{this}.spouse := p$ in method $marry$ might violate the invariant of $\mathbf{this}$ and of all $Person$ objects t where $t.spouse = \mathbf{this}$. Therefore, we impose the following precondition for the field update:

$$\mathbf{assert}\ (\forall\, Person\ t \mid\ t.spouse = \mathbf{this} \bullet\ Person < t.inv\ )\ ;$$

This condition ensures that all potentially affected object invariants are allowed to be violated. Meeting this quantified precondition is easy for a class like $Person$: For any t for which the $Person$ invariants are known to hold, we have $t.spouse.spouse = t$, and so if $t.spouse = \mathbf{this}$, then $\mathbf{this}.spouse = t$. From this observation and the fact that t ranges of non-null references, the $marry$ precondition $\mathbf{this}.spouse = \mathbf{null}$ suffices to establish the quantified precondition for the field update.

Transfer As explained in Section 3, a transfer is essentially an assignment to the $owner$ field of the transferred object. Therefore, the generalized invariants also lead to stronger proof obligations for transfer statements, as illustrated by the following class:

$$\mathbf{class}\ Thief\ \{\ \mathbf{peer}\ Possession\ haul\ ;\ \ldots\ \}$$

Since $haul$ is declared as peer, the class has the implicit invariant

$$\mathbf{invariant}\ haul \neq \mathbf{null}\ \Rightarrow\ haul.owner = owner\ ;$$

which is violated if the object referenced by $haul$ is transferred to another owner. Consequently, such a transfer statement needs an additional precondition that ensures that

possibly affected $Thief$ objects are sufficiently unpacked before a $Possession$ object is transferred.

4.3 Visibility

The above example shows that in the presence of invariants over objects in arbitrary contexts, field updates and transfers have to be guarded by preconditions that assert that all objects that might be affected by the update are sufficiently unpacked. However, such preconditions can be imposed only if the invariants that depend on the updated field are visible in the method that performs the update or transfer. In this subsection, we present the visibility requirements that are necessary to generate the appropriate assertions.

Visibility requirement for declared fields An invariant is called a *visibility-based invariant* if it refers to a field f of an object that is different from $\mathbf{this}$ and that might not be transitively owned by $\mathbf{this}$. Throughout the next few paragraphs, we assume that f is not the $owner$ field; the treatment of $owner$ is discussed below. To guarantee that a visibility-based invariant is visible in every method that might assign to f, we introduce $\mathbf{dependent}$ clauses for field declarations.

If the invariant of a class T contains a field-access expression of the form $\mathbf{this}.g_1.\cdots.g_n.f \ \ (n \geqslant 1)$, where the object $\mathbf{this}.g_1.\cdots.g_n$ might not be (transitively) owned by $\mathbf{this}$, then T must be declared a dependent of f. In our example, $Person$ is declared a dependent of $spouse$, because its invariant refers to $\mathbf{this}.spouse.spouse$, and $spouse$ is not a rep field:

$$\mathbf{peer}\ Person\ spouse\ \mathbf{dependent}\ Person\ ;$$

The dependent-clause allows us to impose the precondition for updates of the $spouse$ field that was presented in the example above.

By the visibility requirement, we can allow more invariants than before. An invariant declared in class T is admissible if for each of its field-access expressions of the form $\mathbf{this}.g_1.\cdots.g_n.f \ \ (n \geqslant 1)$, either g_1 is a rep field and each of the other g_i is a rep or peer field, or T is a dependent of f. Whether an invariant is admissible can be checked syntactically by referring to the $\mathbf{rep}$ and $\mathbf{peer}$ modifiers as well as the $\mathbf{dependent}$ clauses of the involved fields.

As illustrated by the invariants of $Person$ and $Node$ (Fig. 5), visibility-based invariants allow us, for instance, to specify properties of recursive data structures as long as the class that contains the invariant is visible in the classes that declare the fields the invariant depends on. This visibility requirement can easily be met if all involved classes are developed together (in our examples, only one class is involved). However, if a class T comes from a class library, for instance, then the visibility requirement is in general not met; moreover, assuming the library cannot be modified, dependent classes cannot be added to the $\mathbf{dependent}$ clauses of the field declarations in T. In such cases, ownership-based invariants have to be used. That is, if a class S declares a field f of type T and declares an invariant that mentions the fields of f, then f has to be declared with $\mathbf{rep}$. Having to use ownership in this situation does not seem needlessly limiting, but realistic: S implements a new layer on top of class T. Forcing clients

to access (especially modify) the state of lower layers by invoking methods of higher layers is a common design practice.

Visibility requirement for the $owner$ field Ownership transfer is essentially an assignment to the $owner$ field of the transferred object. However, the visibility requirement for ordinary fields is too strong to be useful for $owner$: since $owner$ is a predefined field of class $\mathbf{object}$, the implementor of a class T cannot mention T in the dependent-clause of $owner$. Nevertheless, visibility-based invariants that refer to $owner$ fields are often useful, for instance as implicit invariants for peer fields.

To be able to determine all classes that contain invariants that might be violated by a transfer, we use the $\mathbf{transferable}$ modifier introduced in Section 3 and ensure that all dependent classes are visible in the transferable class. Consequently, the dependent classes are visible in any method that contains a $\mathbf{transfer}$ statement, which allows us to impose an appropriate precondition. We describe this solution in the following.

If the invariant of a class T contains a field-access expression of the form $\mathbf{this}.g_1.\cdots.g_n.owner$ $(n \geqslant 1)$, where the object $\mathbf{this}.g_1.\cdots.g_n$ might not be owned by $\mathbf{this}$, then T must be declared an *owner-dependent* of the static type of g_n. That is, we use the same concept as for dependent-clauses, but instead of listing a dependent class in the field declaration, we specify it in the class of the static type of the field on which $owner$ is accessed. For instance, the implicit invariant for the peer field $spouse$ in class $Person$ refers to $\mathbf{this}.spouse.owner$. Consequently, we have to mention the class that declares the invariant, $Person$, in the owner-dependent declaration of the static type of $\mathbf{this}.spouse$, which also is $Person$. Thus, $Person$ has to contain the declaration $\mathbf{owner\text{-}dependent}$ $Person;$.

Owner-dependent declarations may be specified only for non-transferable classes (*e.g.*, $Person$) and for transferable classes with non-transferable direct superclasses (*e.g.*, $Possession$). That is, transferable classes with transferable superclasses never have any owner-dependent declarations. This restriction allows us to determine all invariants that might be affected by a transfer of the form $\mathbf{transfer}$ x $\mathbf{to}$ $[q, U]$. The classes that declare such invariants are declared owner-dependents of the static type of x, say T, or T's superclasses. Like T and T's superclasses, these owner-dependents are visible in the method that contains the transfer statement. Proper subclasses of T, which might not be visible in this method, are transferable and have a transferable superclass. Therefore, they must not contain owner-dependent declarations and invariants of clients may, in general, not refer to $g.owner$ if the static type of g is a proper subclass of T.

4.4 Proof obligations

The visibility requirement allows us to generalize the proof obligations for field updates and transfers to support both ownership-based and visibility-based invariants.

Precondition for field updates Besides the invariants of x and the invariants of x's transitive owner objects, an update of the field $x.f$ may affect invariants of objects of

the classes in the dependent-clause of f. Therefore, we have to impose the following proof obligation in addition to the assertions described in Section 2.1:

> If a class T is mentioned in the $\mathbf{dependent}$ clause of field f and an invariant of T refers to f such that a T object t depends on $x.f$, then t must be sufficiently unpacked ($T < t.inv$).

This requirement guarantees that visibility-based invariants of an object can be violated only when the object is in a state in which it is known that the invariants may not hold. We formalize this proof obligation in Section 5.

For example, we determine the precondition of the field update $\mathbf{this}.spouse := p$ in method $marry$ (Fig. 6) as follows. The field $spouse$ mentions class $Person$ in its dependent-clause, so we look at the object invariants of $Person$ to determine which invariants may be affected by the $spouse$ assignment. Among the $Person$ invariants, there is one access expression of the form $E.spouse$ where E might not be owned, namely for E being $\mathbf{this}.spouse$. In other words, any $Person$ object t satisfying $t.spouse = \mathbf{this}$ may be affected by the assignment to $\mathbf{this}.spouse$ in $marry$. Thus, our methodology prescribes the following precondition for the field update:

$$\mathbf{assert}\ (\forall\, Person\ t \mid\ t.spouse = \mathbf{this} \bullet\ Person < t.inv\ )\ ;$$

Precondition of transfer The rules for $\mathbf{transferable}$ modifiers and owner-dependent declarations guarantee that, besides the old and new owning objects of x, a transfer of the form $\mathbf{transfer}\ x\ \mathbf{to}\ [q, U]$ can only affect invariants of classes that are mentioned in the owner-dependent declarations of x's static type or of their superclasses. Therefore, we impose a proof obligation that objects of these classes are sufficiently unpacked before the transfer. That is, the following condition has to hold in the pre-state of a transfer of the above form in addition to the requirements presented in Section 3:

> If a class T is mentioned in the owner-dependent declaration in any superclass of x's static type, and the (implicit or explicit) invariant of T refers to $owner$ such that a T object t depends on $x.owner$, then t must be sufficiently unpacked ($T < t.inv$).

* * *

This completes the informal presentation of our methodology. By the visibility requirement, we can allow the invariant of X to refer to objects that are not owned by X without sacrificing modular reasoning. Visibility-based invariants allow one to express properties of data structures locally in a representation class such as $Node$, which simplifies specification and verification of both representation classes and owners. In particular, one can express invariants of object structures even if the owner of the objects is not known. This flexibility is necessary for data structures such as singly-linked lists, where designated owner objects are rather artificial.

5 Technical treatment

In this section, we present the technical treatment of our methodology. That is, we define precisely which invariants are admissible, formalize the assertions for the relevant statements, and prove a soundness theorem.

5.1 Admissible invariants

The invariant of a class C may depend on fields of $\mathbf{this}$ and of objects transitively owned by $\mathbf{this}$, on fields that contain C in their dependent-clause, and on $owner$ fields if C is mentioned in owner-dependent declarations of the static type of the field on which $owner$ is accessed:

Definition 1 (Admissible invariant). *An invariant declaration in class C is admissible if its subexpressions typecheck according to the rules of the programming language and if each of its field-access expressions has one of the following forms:*

1. $\mathbf{this}.g_1.\cdots.g_n.f$*, where either* $n = 0$*, or* g_1 *is a rep field and each of the fields* $g_2, \ldots, g_n$ *is either a rep or a peer field.*
2. $\mathbf{this}.g_1.\cdots.g_n.f$*, where* $n \geqslant 1$*,* f *is different from* $owner$*, and* C *is mentioned in the dependent-clause of* f*.*
3. $\mathbf{this}.g_1.\cdots.g_n.owner$*, where* $n \geqslant 1$ *and* C *is mentioned in an owner-dependent declaration of the type of* g_n*.*
4. $x.f$*, where* x *is bound by a universal quantification of the form*

   $$(\forall\ T\ x \mid\ x.owner = [\mathbf{this}, B] \bullet\ P(x)\ )$$

   and B *is a superclass of* C*.* $P(x)$ *may refer to the identity and the state of* x*, but not to the states of objects referenced by* x*.*

The field f *must not be one of the predefined fields* inv *and* $committed$*.*

The access expression $\mathbf{this}.f$ is a special case of kind 1. Access expressions of kinds 1 and 4 are, for instance, used in the $List$ class shown in Fig. 3. Field-access expressions of kinds 2 and 3 enable visibility-based invariants.

5.2 Proof rules

The methodology presented in this paper does not assume a particular programming logic to reason about programs and specifications. Special rules are required only for those statements that deal with the fields inv and $committed$ ($\mathbf{pack}$, $\mathbf{unpack}$, and field update) as well as $owner$ (object creation and $\mathbf{transfer}$). The rules for pack and unpack statements as well as the specification of $\mathbf{object}$'s constructor are presented in Section 3. We describe the rules for the remaining statements in the following.

Field updates The rule for field updates was explained in Section 4.4. More formally, a field update of the form $x.f := e$ is guarded by the following preconditions:

1. $\mathbf{assert}\ x \neq \mathbf{null}\ \wedge\ F < x.inv$;
   where F is the class in which f is declared.
2. For each class T mentioned in the dependent-clause of f, and for each access expression $\mathbf{this}.g_1.\cdots.g_n.f$ of kind 2 (and not of kind 1) in an invariant declared in T: $\mathbf{assert}\ (\forall\ T\ t \mid\ t.g_1.\cdots.g_n = x \bullet\ T < t.inv\ )$;.

The first precondition is identical to the methodology with ownership-based invariants only. It guarantees that x's transitive owner objects are sufficiently unpacked. Preconditions of the second kind are necessary to handle invariants with field-access expressions of kind 2 in Def. 1.

Transfer The rule for ownership transfer is analogous to field updates, but refers to owner-dependent declarations instead of dependent-clauses: A transfer statement of the form $\mathbf{transfer}\ x\ \mathbf{to}\ [q, U]$ is guarded by the following preconditions:

1. $\mathbf{assert}\ x \neq \mathbf{null} \wedge x.inv = \mathbf{object}$;
2. $\mathbf{assert}\ x.owner.obj \neq \mathbf{null}\ \Rightarrow\ x.owner.typ < x.owner.obj.inv$;
3. $\mathbf{assert}\ q \neq \mathbf{null}\ \Rightarrow\ U < q.inv$;
4. For each class T mentioned in an owner-dependent declaration in any superclass of x's static type, and for each access expression $\mathbf{this}.g_1.\cdots.g_n.owner$ of kind 3 (and not of kind 1) in an invariant declared in T:

$$\mathbf{assert}\ (\forall\ T\ t \mid\ t.g_1.\cdots.g_n = x \bullet\ T < t.inv\ )\ ;$$

The first three preconditions were also part of the methodology with ownership-based invariants only. They ensure that both x's old and new owner objects are sufficiently unpacked. Preconditions of the last kind are necessary to handle invariants with field-access expressions of kind 3 (Def. 1), which occur for instance in the implicit invariants for peer fields.

5.3 Soundness

For our methodology, soundness means that the inv field of an object x correctly reflects which invariants of x can be assumed to hold. In this subsection, we formalize and prove this property for well-formed programs. A program **P** is well-formed if **P** is syntactically correct, type correct, and **P**'s invariants are admissible (see Def. 1).

Theorem 1 (Soundness theorem). *In each reachable execution state of a well-formed program, the following program invariant holds:*

$$(\forall\, x, T \bullet\ x.inv \leqslant T\ \Rightarrow\ Inv_T(x)\ )$$

where $Inv_T(x)$ expresses that x satisfies all (explicit and implicit) invariants declared in class T.

Soundness proof Because of limited space, we present the proof of a simplified theorem here that assumes that all field-access expressions of admissible invariants (Def. 1) refer to fields of $\mathbf{this}$ or a bound variable, or to fields of objects directly referenced by $\mathbf{this}$. That is, for field-access expressions of kinds 1 to 3, we assume $n \leqslant 1$. A generalization of the proof is straightforward, but requires several auxiliary lemmas about transitive ownership we cannot present here.

The proof runs by induction over the sequence of states of a program **P**. The induction base is trivial. For the induction step, only the statements that create objects or manipulate fields of objects are interesting. We omit all trivial cases for brevity.

Object creation Creation of a new object x does not change the values of fields of existing objects. Since a precondition of the operation is that any given owning object of x is sufficiently unpacked, we only have to show that the property holds for x itself, which is a direct consequence of the fact that $x.inv = \mathbf{object}$ and class $\mathbf{object}$ does not have invariants.

Pack A pack statement changes the inv field of the packed object as well as the $committed$ fields of objects directly owned by that object, but nothing else. Since invariants must not refer to inv or $committed$ fields (see Def. 1), the value of $Inv_T(x)$ cannot be changed by a pack statement. The value of $x.inv \leqslant T$ is only changed by the statement $\mathbf{pack}\ x\ \mathbf{as}\ T$. However, the precondition of the pack statement checks that $Inv_T(x)$ holds. Therefore both sides of the implication yield $true$ after the statement.

Unpack Like pack statements, unpack statements only change inv and $committed$ fields, which implies that $Inv_T(x)$ is not affected by any unpack statement. The value of $x.inv$ after the statement is a direct superclass of the value before the statement. Thus, the value of $x.inv \leqslant T$ might only be changed from $true$ to $false$. That is, the implication still holds after the unpack statement.

Field update Let f be a field declared in a class F and consider the effect of an update $y.f := e$ on $Inv_T(x)$ for some x and T. In particular, we show that if $Inv_T(x)$ contains an access expression that denotes $y.f$, then x is sufficiently unpacked: $T < x.inv$ (that is, the left side of the implication is $false$). For this proof, we only only need to consider access expressions in the invariants that end with dereferencing f. Access expressions that mention f somewhere in the middle contain as a subexpression an access expression that ends with f. Following (the simplified) Def. 1, we consider the following cases:

1a. An invariant of T refers to $\mathbf{this}.f$ and $x = y$: The precondition of the field update requires $F < x.inv$. Since T is a subclass of F (otherwise the expression $x.f$ would not typecheck), we get $T < x.inv$.

1b. An invariant of T refers to $\mathbf{this}.g.f$, where g is a rep field declared in a superclass S of T, and $x.g = y$: From the precondition of the update of $y.f$, we know that y is not consistent. Since $x.g = y$ and g is a rep field, x must be unpacked beyond S: $S < x.inv$. (The definition of the unpack operation guarantees that an owner object x is unpacked beyond the owner type S before an object owned by $[x, S]$ can be unpacked. The pack operation guarantees that objects are packed in the reverse order.) Since T is a subclass of S, we have $T \leqslant S < x.inv$.

2. An invariant of T refers to $\mathbf{this}.g.f$, $x.g = y$, and T is mentioned in f's dependent-clause: The field update has the precondition

$$\mathbf{assert}\ (\forall\ T\ t \mid\ t.g = y \bullet\ T < t.inv\ )\ ;$$

Instantiating t with x, we have $T < x.inv$.

3. An invariant of T refers to $\mathbf{this}.f.owner$ and $x = y$: Since we only have to consider access expressions that end with dereferencing f, there is nothing to be shown for this case. $\mathbf{this}.f.owner$ has $\mathbf{this}.f$ as a subexpression, which is handled in Case 1a.

4. An invariant of T refers to $o.f$, where o is a variable bound by quantification, $o = y$, $o.owner = [x, S]$, and $T \leqslant S$: Analogously to Case 1b, y is not consistent, and y's owner object, x, must be unpacked beyond the owner type, S: $S < x.inv$. Since T is a subclass of S, we have $T \leqslant S < x.inv$.

Transfer Consider the statement $\mathbf{transfer}\ y\ \mathbf{to}\ [q, U]$. This transfer is essentially an update of $y.owner$. Therefore, the proof for a transfer is similar to the proof for field updates: Cases 1a, 1b, and 4 are analogous. Case 2 for transfers is analogous to Case 3 of field updates and vice versa. However, Case 3 for transfers has to refer to the owner-dependent declaration of the static type of $\mathbf{this}.g$ instead of the dependent-clause of f. Moreover, all cases have to consider both the old and the new owner of y. However, since both owner objects are sufficiently unpacked before the transfer, the arguments of the cases for field updates apply as well to the new owner object of the transferred object. □

6 Usability

In this section, we discuss the expressiveness and practicability of our methodology.

Expressiveness The methodology presented here can express properties of many interesting implementations. Ownership-based invariants allow us to specify properties of the internal representation of an object structure in a modular way. By supporting quantifications over all objects owned by an object, we can handle complex object structures such as the union-find data structure, where not all constituent objects are reachable from the owning object. Visibility-based invariants enable one to declare invariants locally in classes of constituent objects, which simplifies specifications and proofs and, in particular, allow us to handle data structures that do not have an explicit owner.

Although we do not restrict aliasing, our methodology requires that modifications of objects always be initiated by their owner objects since the owner object has to be unpacked before the modification. Therefore, data structures like collections with iterators are difficult to handle with ownership-based invariants; essentially, an iterator needs to arrange to unpack the collection before it can modify the collection's state. In our $List$ example, we could use visibility-based invariants for a class $Iterator$ if $Node$ and $Iterator$ are mutually visible. But either the list or its owner would have to know all iterators of the list to be able to unpack them all before the list is modified. To provide better support for such patterns, one might consider generalizing our methodology to allow multiple owners for each object and adding support for packing and unpacking all owners of an object simultaneously.

In this paper, we omitted arrays for brevity. The treatment of arrays is analogous to other objects. In particular, arrays can have (implicit) invariants specifying the owner of their elements. So far, our methodology does not support static fields. However, static fields can be treated as fields of class objects, which allows visibility-based invariants to express properties of global state.

Specification support Due to the semantics of $\mathbf{modifies}$ clauses, methods can allocate and modify new objects without explicitly specifying these modifications. However, if the caller has no knowledge about new objects and their consistency, it is in general not possible to reason about invariants that quantify over all objects owned by a certain owner. To deal with this problem, the specifications in our $List$ example contain

ensures clauses that state that all newly allocated objects are consistent upon termination of a routine (see Fig. 5). Instead of writing such **ensures** clauses, it would be helpful to provide a designated **expands** clause that specifies the owners of objects allocated by a routine. Analogous with the rule for **modifies** clauses, if the owning object is newly allocated or on entry is committed, then the **expands** clause would not need to mention the owner. For instance, the clause **expands** ow for $Node$'s constructor would say that the constructor does not create objects for owners other than ow, which would simplify the corresponding **ensures** clause.

In this paper, we use dependent-clauses to check the visibility requirement. Such clauses make potential dependencies explicit, but lead to additional specification overhead. For languages that provide modules with acyclic import, dependent-clauses are not necessary. A tool could infer the dependent-clauses within one module. Inter-module dependencies violate the visibility requirement and are, thus, forbidden. This approach is, for instance, taken in Müller's thesis [30].

Many of the specification constructs we've discussed are often used in certain stylized forms. For example, methods often require their parameters to be consistent and uncommitted, pack and unpack statements tend to occur in pairs in the bodies of public methods, and the owner parameter passed to constructors often mention **this** and the enclosing class. We'd like to experiment with useful defaults that reduce the number of explicit specifications are needed in a program.

Tool support Although we consider our work a significant improvement in the treatment of object invariants, reasoning is still tedious and appropriate tool support is indispensable. One of the key considerations for the design of our methodology was to avoid reachability predicates in proof obligations since existing theorem provers can have trouble handling such recursive predicates automatically (*cf.* [19]). Specifications such as *all nodes reachable from* $\mathbf{this}.head$ *are owned by* $[\mathbf{this}, List]$ can be avoided by quantification in ownership-based invariants (see Fig. 3) or by visibility-based invariants (see Fig. 5).

We plan to implement our methodology as part of the .NET program checker Boogie at Microsoft Research and use this implementation for non-trivial case studies. Applying the Extended Static Checker for Java [16] to a special encoding of the examples used in this paper lead to promising results: all proof obligations except those for **modifies** clauses, which are not checked by ESC/Java, were proved automatically.

7 Related work

In this section, we discuss papers from the large literature on invariants that are directly related to invariants for object structures. A more detailed discussion of related work is found in Müller's thesis [30], and we gave a more detailed comparison with two previous methodologies in Section 2.

Classical proof systems for invariants such as Meyer's work [29, 28] or the approach of Liskov, Wing, and Guttag [27, 26] are not sound if invariants express properties of more than one object. They require exported methods to preserve the invariant only of the current object or of all objects of the enclosing class, neither of which is sufficient

for general object structures with aliasing [32]. Especially, behavioral subtyping [22, 13] is necessary but not sufficient to guarantee modular soundness for invariants over object structures. Other specification languages such as JML [21, 20] or several Larch languages [17] permit invariants over object structures but do not provide a sound modular proof system.

Huizing and Kuiper [18] present a proof system that supports invariants for object structures. Invariants are analyzed syntactically to determine which methods of a program might violate which invariants. However, without appropriate restrictions, this analysis is not modular.

Banerjee and Naumann [1] consider what it means, formally, for the exported interface of a class to be independent of the implementation of the class, which may rely on object invariants. They use ideas from separation logic and permit the heap to be partitioned in a flexible way (for example, constituent objects need not be reached from the owner). Their semantic results are sound even in the presence of call-backs, but just how one goes about establishing the antecedents of their theorems is mostly left unaddressed.

The approach presented in our paper is based on recent work by Barnett *et al.* [2]. We replaced the static declaration of components by a dynamic encoding of ownership, which enables invariants over dynamically growing and shrinking object structures such as linked lists. In addition, our approach supports visibility-based invariants, which can simplify specifications and proofs, and which allow us to handle object structures that do not have explicit owner objects.

The treatment of visibility-based invariants was influenced by Müller's thesis [30]. Müller uses a visible state semantics for invariants that requires invariants of relevant objects to hold in pre- and postconditions of all exported methods, whereas our methodology allows invariants to be violated as long as such violations are made explicit by the inv field. This type-valued inv field especially provides a better handling of inheritance [2]. Müller's thesis supports invariants over so-called abstract fields in a sound way, which we consider future work for the methodology presented here. Müller's approach has been applied to more restricted invariants [32] and to the treatment of **modifies** clauses [33].

The programming model supported by Müller's universe type system corresponds to the restrictions used in this paper: Objects can be freely aliased, but modifications of objects have to be initiated by owners. Besides universes, several other type systems have been proposed to express ownership statically [4–9]. Some ownership type systems use ownership parameters to express ownership relations like the ones we specify with **rep** and **peer** tags. In contrast to our work, these type systems restrict aliasing of objects and do not support ownership transfer. Ownership type systems guarantee the ownership relation in all execution states, whereas the ownership relations specified by our object invariants (including the implicit object invariants for **rep** and **peer** fields) are conditions that need not always hold. This dynamic encoding simplifies transfer, but makes the soundness proof more difficult. Clarke and Wrigstad [10] combine ownership types with externally unique references to permit transfer. Fähndrich and DeLine [15] present a type system with linearity for checking interface protocols of objects. Their adoption and focus statements provide a controlled way of creating aliases and access-

ing aliased objects, loosening the rigid uniqueness requirements imposed by linear type systems.

Barnett and Naumann [3] extend the work presented in our paper. In their encoding, each object X maintains a (possibly abstract) set of objects whose visibility-based invariants depend on fields of X. This set can be used to establish the preconditions for updating X's fields more easily than in our approach. Another novelty of Barnett and Naumann's paper is update guards. An update guard abstracts the weakest precondition for a field update, to enable the update without unpacking potentially affected objects or exposing internal state in that precondition.

Like our methodology, the work by Leino *et al.* [23, 24, 11] imposes visibility requirements to enable a modular sound treatment of abstract fields. However, their work is not based on the notion of ownership, which leads to more complicated requirements for specifying properties over object structures and makes the soundness proof difficult. The Extended Static Checker for Modula-3 [12] uses the technique of Leino and Nelson [24] to reason about validity of object structures by defining a boolean abstract field $valid$ to represent validity. Usage of this field in specifications is similar to our inv field.

The Extended Static Checker for Java [16] uses heuristics to determine which object invariants to check for method invocations. Described in detail in the ESC/Java User's Manual [25], these heuristics are a compromise between flexibility and likelihood of errors and do not guarantee soundness.

Our technique requires programmers to specify invariants as well as rep and peer annotations explicitly. Tools such as Daikon [14] could be used to guess possible object invariants automatically and then check them by our methodology.

8 Conclusions

We presented a methodology for specifying and reasoning about object invariants. Our solution allows one to express properties of object structures without restricting aliasing. A combination of ownership- and visibility-based invariants provides enough expressiveness to handle non-trivial implementations such as (mutually) recursive data structures and re-entrant method calls. Inheritance is fully supported. The methodology is modular and proven to be sound.

As future work, we plan to generalize our methodology to allow objects to have multiple owners, which is, for instance, necessary for certain implementations of collections with iterators. Invariants over abstract fields are useful to describe properties of data structures without referring to their concrete implementation. We plan to adapt our previous work on abstract fields [30, 23] to the methodology presented here.

Acknowledgments We are grateful to Mike Barnett, John Boyland, Rob DeLine, Manuel Fähndrich, Bertrand Meyer, Dave Naumann, and Wolfram Schulte for helpful discussions on invariants and ownership. We especially thank Dave Naumann, who suggested we separate the concepts of peer fields and visibility-based invariants, which in a previous version had been entangled. We also thank the referees for helping improve the presentation.

References

1. Anindya Banerjee and David A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. Manuscript available on `http://guinness.cs.stevens-tech.edu/~naumann/publications/`, December 2002.
2. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 2004. To appear.
3. Mike Barnett and David Naumann. Friends need a bit more: Maintaining invariants over shared state. In *Mathematics of Program Construction*, Lecture Notes in Computer Science. Springer-Verlag, 2004. To appear.
4. Boris Bokowski and Jan Vitek. Confined types. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '99)*, volume 34, number 10 in *SIGPLAN Notices*, pages 82–96. ACM, October 1999.
5. Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002*, volume 37, number 11 in *SIGPLAN Notices*, pages 211–230. ACM, November 2002.
6. Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, volume 38, number 1 in *SIGPLAN Notices*, pages 213–223. ACM, January 2003.
7. Dave Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, 2001.
8. Dave G. Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002*, volume 37, number 11 in *SIGPLAN Notices*, pages 292–310. ACM, November 2002.
9. Dave G. Clarke, John. M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98)*, volume 33, number 10 in *SIGPLAN Notices*, pages 48–64. ACM, October 1998.
10. Dave G. Clarke and Tobias Wrigstad. External uniqueness is unique enough. In Luca Cardelli, editor, *ECOOP 2003—Object-Oriented Programming, 17th European Conference*, volume 2743 of *Lecture Notes in Computer Science*, pages 176–200. Springer, 2003.
11. David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. Research Report 156, Digital Equipment Corporation Systems Research Center, July 1998.
12. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.
13. Krishna Kishore Dhara. Behavioral subtyping in object-oriented languages. Technical Report 97-09, Iowa State University, May 1997.
14. Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, 2001.
15. Manuel Fähndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37, number 5 in *SIGPLAN Notices*, pages 13–24. ACM, May 2002.

16. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37, number 5 in *SIGPLAN Notices*, pages 234–245. ACM, May 2002.
17. John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.
18. Kees Huizing and Ruurd Kuiper. Verification of object-oriented programs using class invariants. In Tom Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 208–221. Springer-Verlag, 2000.
19. Rajeev Joshi. Extended static checking of programs with cyclic dependencies. In James Mason, editor, *1997 SRC Summer Intern Projects*, Technical Note 1997-028. Digital Equipment Corporation Systems Research Center, 1997.
20. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.
21. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06v, Iowa State University, Department of Computer Science, May 2003. See www.jmlspecs.org.
22. Gary T. Leavens and Krishna Kishore Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*. Cambridge University Press, 2000.
23. K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995.
24. K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.
25. K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical Note 2000-002, Compaq Systems Research Center, October 2000.
26. Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Electrical Engineering and Computer Science Series. MIT Press, 1986.
27. Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), 1994.
28. Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
29. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition edition, 1997.
30. Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. PhD thesis, FernUniversität Hagen.
31. Peter Müller and Arnd Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, FernUniversität Hagen, 2001.
32. Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. Technical Report 424, Department of Computer Science, ETH Zurich, 2003.
33. Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15:117–154, 2003.