

Universes: Lightweight Ownership for JML

Werner Dietl and Peter Müller, ETH Zurich, Switzerland

Object-oriented programs with arbitrary object structures are difficult to understand, to maintain, and to reason about. Ownership has been applied successfully to structure the object store and to restrict how references can be passed and used.

We describe how ownership relations can be expressed in the Java Modeling Language, JML. These ownership specifications can be checked by standard verification techniques, runtime assertion checking, ownership type systems, or combinations of these techniques. We show that the combination of the lightweight Universe type system and JML specifications is flexible enough to handle interesting implementations while keeping the annotation and checking overhead small.

The Universe type system has been implemented in the JML compiler. This integration enables the application of ownership-based verification techniques to programs specified in JML.

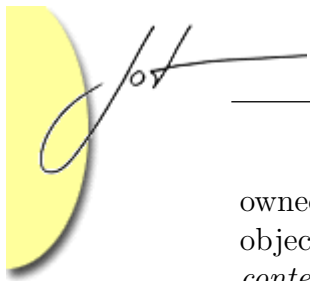
1 INTRODUCTION

In object-oriented programs, an object can potentially reference any other object in the object store and read and modify its fields through direct field accesses or through method calls. Such programs with arbitrary object structures are difficult to understand, to maintain, and to reason about. In particular, modular verification of functional correctness properties typically requires control of how references are passed around and what operations can be performed on references.

For instance, to prove that a list data structure maintains an invariant that the list is sorted, one has to show that all methods that can modify the internal representation of the list, say, an underlying array, preserve the invariant. This can be achieved more easily by enforcing that only certain objects such as the list object can modify the array directly whereas all other objects have readonly access to the array or no direct reference to the array at all.

Ownership has been applied successfully to structure the object store and to restrict reference passing and the operations that can be performed on references. In particular, ownership allows one to confine an object inside a data structure and to prevent representation exposure through leaking [NVP98].

The restrictions on references simplify reasoning about programs: they enable modular verification [LM04, Mül02, MPHL03, MPHL04], facilitate thread synchronization [BLR02], and allow programmers to exchange internal representations of data structures [BN02]. Ownership organizes objects into *contexts*: Each object is



owned by at most one other object, called its *owner*. A context is the set of all objects with the same owner. The set of objects without an owner is called the *root context*. The contexts of a program execution form a tree, where the context of all objects with owner X is a child of the context containing X . The context tree is rooted in the root context.

Most ownership models enforce the *owner-as-dominator* property: All reference chains from an object in the root context to an object X in a different context go through X 's owner [CPN98, Cla01]. This restriction allows an owner object to control how the objects it (transitively) owns are accessed. For the verification of functional correctness properties such as object invariants, a weaker ownership model suffices: an object X can be referenced by any other object, but reference chains that do not pass through X 's owner must not be used to modify X [LM04, Mül02]. This model distinguishes between *readwrite* and *readonly* references, and enforces the owner-as-dominator property only on readwrite references. Owners can control modifications of owned objects, but not read access. We call this property *owner-as-modifier*.

Both kinds of ownership models enable modular verification. In the above example, the list object would own the underlying array, thereby preventing other objects from modifying the array without going through the list interface. This allows the list to maintain its invariant.

Ownership properties can be checked statically by type systems. Most existing work focuses on parametric ownership type systems that enforce the owner-as-dominator property [CD02, CPN98, PNCB03]. Although these type systems can express and check the ownership properties of many interesting programs, there are several common implementation patterns that do not follow the owner-as-dominator structure such as collections with iterators and producer/consumer with a shared buffer. The ownership type systems by Boyapati and Clarke [Boy04, BLS03, Cla01] weaken the owner-as-dominator property by allowing instances of inner classes to access the representation of the instance of the outer class they are associated with. Thereby, they can handle iterators, but not more general forms of sharing. While parametric ownership type systems describe ownership properties accurately and guarantee a strong type invariant, ownership parametricity increases the complexity of the type system and the annotation overhead.

This paper proposes an alternative approach to specifying and checking ownership:

1. We focus on the owner-as-modifier property. This property has been shown to enable the modular verification of functional correctness properties [BDF⁺04, LM04, Mül02] and can handle more implementations than owner-as-dominator.
2. To check ownership statically, we use the Universe type system [Mül02, MPH01], which requires less annotation overhead than other ownership type systems.



3. The Universe type system does not use ownership parametricity. To compensate for the resulting weaker static guarantees, we combine ownership type annotations with specifications in the Java Modeling Language, JML [LBR99, LBR04]. JML can be used to specify fine-grained ownership information that cannot be expressed by the Universe type system. Such specifications can be checked at runtime or proved statically by standard verification techniques.

The main contribution of this paper is the integration of the Universe type system and JML. This integration serves two purposes: (1) It allows one to apply ownership-based verification techniques to programs specified in JML. (2) It combines the advantages of type checking with the flexibility of interface specifications and runtime assertion checking or verification. We illustrate the expressiveness of the combination of the Universe type system and JML by examples that are difficult to handle in other ownership type systems.

To focus on these contributions, we consider only a subset of Java. Throughout the paper, we omit the treatment of static class members, exceptions, inner classes, generics, overloading, and reflection. Our implementation handles these features, except for generics and reflection.

Outline. In the next section, we show how ownership properties can be expressed in JML and checked by verification techniques or runtime assertion checking. Section 3 presents an introduction to the Universe type system. Section 4 illustrates the expressiveness of the combination of a lightweight owner-as-modifier type system and interface specifications by two examples. We describe our implementation of the Universe type system in JML in Section 5. Related work is discussed in Section 6.

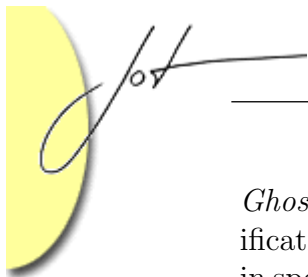
2 SPECIFYING OWNERSHIP PROPERTIES IN JML

In this section, we show how ownership properties can be expressed by JML specifications. We use invariants, requires clauses, and ensures clauses to describe the ownership relations between objects. Both static verification and runtime assertion checking can be used to check that a program meets its ownership annotations and that it satisfies the owner-as-modifier property. We present a type system to check certain ownership properties syntactically in the next section.

Ownership Encoding

To encode the ownership relation, each object stores a reference to its owner or `null` if the object belongs to the root context. For this purpose, we use the ghost field `owner` declared in class `Object`:

```
/*@ ghost public Object owner; @*/
```



Ghost fields are specification-only fields that can be read and updated in JML specifications, but are not accessible for the Java code. (JML specifications are enclosed in special comment delimiters `/*@ ... @*/` or on lines preceded by `//@`.) We call the encoding of the ownership relation by a ghost field a *dynamic encoding* as opposed to the static encoding by a type system.

The `owner` field of an object is set when the object is created. The owner of a new object is specified in the `new` expression. To simplify the syntax, which is described in the next subsection, we require that the new object is owned by `this` or is in the context that contains `this`. So far we found only one practical example that was ruled out by this restriction: a collection cannot create an iterator in the context of an arbitrary client. Instead, the iterator has to be created locally in the context that contains the collection, which prevents clients in other contexts from modifying the iterator. Generalizing object creation to arbitrary owners is straightforward, but requires that the owner of the new object is passed as additional (ghost) parameter to constructors [LM04].

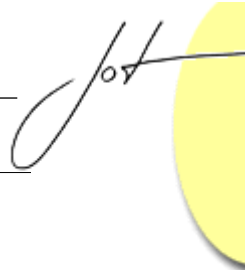
By using a ghost field to encode ownership, ownership transfer can easily be expressed by setting the `owner` field to a new value [LM04]. In this paper, we do not consider ownership transfer because it is difficult to handle by ownership type systems [CW03]. That is, the `owner` field of an object cannot be changed after the object is created. This can be statically checked by disallowing JML's `set` operation for `owner`. As future work, we plan to combine the Universe type system with uniqueness to support ownership transfer.

The references stored in the `owner` fields represent a binary relation on objects. Its reflexive, transitive closure is the ownership relation. Our encoding guarantees that the ownership relation is a tree order: (1) Each object has at most one owner; (2) The ownership relation is acyclic, because the owner of an object X is allocated before X is created and cannot be changed afterwards.

Ownership Modifiers

The `owner` field can be mentioned in method specifications and invariants. Class `Node` in Fig. 1 implements the nodes of a doubly-linked list. Its invariants use `owner` to express that each `Node` object is in the same context as its predecessor and its successor.

To simplify such specifications, we use syntactic abbreviations for the most common ownership relations. The *ownership modifier* `peer` can be used to express that a program element yields a reference to an object which has the same owner as the `this` object. Similarly, `rep` expresses that the object is owned by `this`. The `peer` and `rep` modifiers can be used in the declaration of instance fields and in the declaration of the formal parameters and the result of instance methods, provided that the field, parameter, or method result is of a reference type. Since we do not discuss static methods in this paper, there is always a current receiver object `this`.



```

class Node {
  Node prev, next;
  Object elem;
  //@ invariant prev == null || prev.owner == this.owner;
  //@ invariant next == null || next.owner == this.owner;
}

```

Figure 1: The class for the nodes of a doubly-linked list. The invariant expresses that a node is in the same context as its neighbors.

An extension to static methods is straightforward [Mül02].

Ownership modifiers are also used to specify the owner of a newly created object. The expression `new /*@ peer @*/ T()` creates a new `T` object and sets its `owner` field to `this.owner`. Analogously, `new /*@ rep @*/ T()` creates a new object owned by `this`.

Besides acting as syntactic shorthands, ownership modifiers are useful to check syntactically the constraints of ownership-based verification techniques, for instance, that the invariant of an object `X` refers only to fields of `X` and fields of objects transitively owned by `X` [LM04, MPHL04]. Moreover, they are key to ownership type checking, see Sec. 3.

Desugaring. The meaning of ownership modifiers is defined by desugaring them into invariants, requires clauses, and ensures clauses. The modifiers in the instance field declarations

```

/*@ peer @*/ T peerField;
/*@ rep @*/ T repField;

```

give rise to the implicit invariants

```

//@ invariant peerField == null || peerField.owner == this.owner;
//@ invariant repField == null || repField.owner == this;

```

Analogously, ownership modifiers of formal parameters are desugared into requires clauses and modifiers of method results lead to ensures clauses. Fig. 2 shows an alternative implementation of class `Node` with an ownership modifier, which is desugared into the invariants shown in Fig. 1.

The desugaring of ownership modifiers in `new` expressions is a bit more difficult. Conceptually, the owner of a new object is passed to the constructor as additional parameter, which is assigned to `owner` by `Object`'s default constructor. Therefore, an implicit ensures clause can be added to each constructor expressing that the new object has the specified owner. To avoid the extra parameter for constructors and the corresponding changes to API classes such as `Object`, we took a different approach in our implementation, which is described in Sec. 5.

```

class Node {
  /*@ peer @*/ Node prev, next;
  Object elem;
}

```

Figure 2: Class `Node` with an ownership modifier, which is desugared into the invariants shown in Fig. 1.

Checking Ownership Properties

In this subsection, we show how static verification or runtime assertion checking can be used to check whether an implementation satisfies the specified ownership relation and whether it adheres to the owner-as-modifier property.

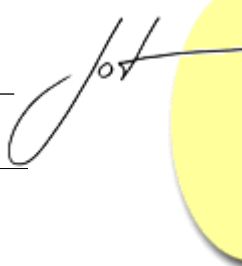
Ownership Relation. The ownership relation is expressed using the ghost field `owner` in invariants and method specifications. This allows one to apply standard verification techniques to prove statically that a program satisfies its ownership specification. Alternatively, the assertions can be evaluated at runtime, but we have not yet implemented the checks in the JML runtime assertion checker [Che03, CL02].

We illustrate the checking of ownership properties by the example in Fig. 3. Class `LinkedList` implements the head of a doubly-linked list, which owns the list nodes. Therefore, the `first` field is declared with the `rep` modifier.

To show that the constructor is correct, one has to check that it establishes the implicit invariant given by the `rep` declaration. The field `first` is initialized with a new `Node` object. After the assignment, we have `first.owner == this`. (For static verification, we get this property from the implicit `ensures` clause of the `Node` constructor.) Therefore, the invariant holds in the poststate of the constructor.

Method `capture` violates the implicit invariant by assigning an object in the same context as `this` to `first`. This bug would become manifest during runtime checking when the implicit invariant is checked before the method terminates. The bug can also be detected by static verification: By the explicit and implicit `requires` clauses, we may assume `n != null` and `n.owner == this.owner`, resp. Therefore, after the assignment we have `first.owner == this.owner` and, since `this != this.owner` (the ownership relation is acyclic), we have `first.owner != this`. That is, the invariant is not preserved by the method. This demonstrates how the implicit invariant can be checked.

Owner-as-Modifier. A fundamental difference between owner-as-dominator and owner-as-modifier is that the owner-as-dominator property restricts *where* references are allowed to point to, whereas owner-as-modifier models allow references to point to objects in arbitrary contexts, but restrict *how* references can be used. Therefore,



```

public class LinkedList {
  /*@ spec_public rep @*/ Node first;

  public LinkedList(Object e) {
    first = new /*@ rep @*/ Node();
    first.elem = e;
  }

  /*@ requires n != null;
  public void capture(/*@ peer @*/ Node n) {
    first = n; // illegal: violates implicit invariant
  }

  /*@ requires l != null && l != this;
  @ requires first != null && l.first != null;
  @*/
  public void exchangeFirst(/*@ peer @*/ LinkedList l) {
    Object tmp = first.elem;
    first.elem = l.first.elem;
    l.first.elem = tmp; // illegal: violates owner-as-modifier
  }

  // other methods omitted
}

```

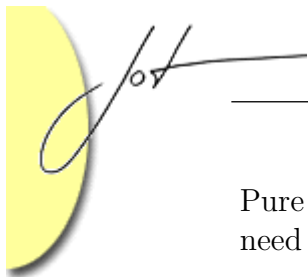
Figure 3: Implementation of a doubly-linked list. The incorrect methods `capture` and `exchangeFirst` violate the implicit ownership invariant and the owner-as-modifier property, resp. These errors can be detected by static verification or runtime assertion checking.

owner-as-dominator requires checks whenever a reference is potentially passed to an object in another context (for instance, by field accesses or method calls), whereas owner-as-modifier requires checks whenever an object is potentially modified.

The state of an object X can be modified either directly by assigning to the fields of X or indirectly by calling a method on X that performs the modification. The owner-as-modifier property can be guaranteed by checking that these operations are performed only by objects in the context that contains X or by X 's owner.

In JML, checks or proof obligations within method bodies are expressed by `assert` clauses. To enforce the owner-as-modifier property, we guard each field update of the form $X.f = e$ and each call $X.m(\dots)$ of a non-pure method m by the following assertion:

```
assert X.owner == this.owner || X.owner == this;
```



Pure methods do not change the state of allocated objects. Therefore, their use need not be restricted.

Like for all other ownership specifications, static verification or runtime assertion checking can be used to check the above assertions. In the `LinkedList` example, the constructor and method `capture` can be checked easily. They update only fields of `this`. Thus, the first disjunct of the assertion is met trivially.

Method `exchangeFirst` violates the owner-as-modifier property by directly modifying the internal representation of another list, `l`, without calling a method on `l`. This violation is detected by checking the assertion for the update `l.first.elem = tmp`. By `l`'s implicit invariant, we have `l.first.owner == l`, by the acyclicity of the ownership relation, we have `l != l.owner`, and by the implicit `requires` clause, we have `l.owner == this.owner`. This implies `l.first.owner != this.owner`, that is, the first disjunct of the assertion does not hold. The second disjunct of the assertion, `l.first.owner == this`, does not hold because (1) `l.first.owner == l` (by `l`'s implicit invariant) and (2) `l != this` (by the explicit `requires` clause).

Since ownership can be expressed by standard JML constructs, a large number of techniques and tools for JML [BCC⁺03, BRL03, Che03, CK04, FLL⁺02, Jac04, JP01] can be used to reason about ownership properties. In the next section, we show how these properties can be checked syntactically by the Universe type system.

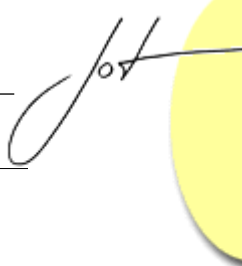
3 THE UNIVERSE TYPE SYSTEM

Ownership annotations lead to a high number of assertions that have to be proved or checked at runtime. The Universe type system can check most of these assertions syntactically, thereby reducing the verification or runtime checking overhead significantly. In this section, we give an informal introduction to the Universe type system. A formalization and a type safety proof is found in our earlier work [Mül02, MPH01].

Types and Subtyping

The Universe type system distinguishes three kinds of references: (1) references between objects in the same context (*peer* references), (2) references from an object `X` to an object directly owned by `X` (*rep* references), and (3) references between objects in arbitrary contexts. According to the owner-as-modifier property, references of the third kind must not be used to modify the referenced object since the reference is not guaranteed to come from the owner or a peer object of the modified object. Hence, we call these references *readonly* references.

The Universe type system uses the ownership modifiers `peer` and `rep` plus the additional modifier `readonly` to classify references into the three kinds described above. In contrast to the dynamic encoding, where arbitrary ownership is expressed by not adding an ownership modifier to a declaration, the Universe type system uses



```
class Node {
  /*@ peer */ Node prev, next;
  /*@ readonly */ Object elem;
}
```

Figure 4: Class `Node` with Universe types.

the `readonly` modifier to make explicit that a reference may point to any context (kind 3). Having an explicit `readonly` is useful to record design decisions. It also allows us to use `peer` as a default when an ownership modifier is omitted (see Sec. 5).

Types. In contrast to the dynamic ownership encoding, the Universe type system requires that an ownership modifier is specified for each expression that evaluates to a reference at runtime. Therefore, we associate ownership modifiers with reference types.

The types of the Universe type system comprise Java’s primitive types, class and interface types, and array types. Class and interface types are pairs of an ownership modifier (`peer`, `rep`, or `readonly`) and a class or interface name. For instance, the type `rep T` is the type of references pointing to objects of class or interface `T` owned by `this`.

Array types have two ownership modifiers, one for the array object and one for the elements of the array. (The second modifier is omitted for arrays of primitive types such as integer arrays.) The type `rep readonly T[]` is the type of an array owned by `this`, whose elements are instances of `T` in arbitrary contexts. All array objects of a multi-dimensional array belong to the same context.

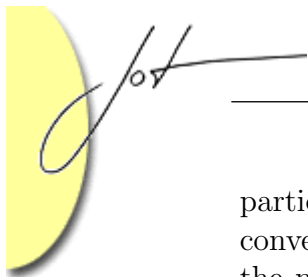
According to their (first) ownership modifier, we call reference types *peer*, *rep*, and *readonly* types, resp.

Fig. 4 shows the `Node` class with Universe types. The `readonly` modifier expresses that the elements of the list can belong to arbitrary, even different, contexts.

Subtyping. The subtype relation on Universe types follows the subtype relation in Java: Two `peer`, `rep`, or `readonly` types are subtypes if the corresponding classes or interfaces are subtypes in Java. In addition, every `peer` and `rep` type is a subtype of the `readonly` type with the same class, interface, or array element type, because it is more specific in terms of the context information it conveys.

The Universe type system has covariant array subtyping. That is, two array types with the same ownership modifier are subtypes if their element types are subtypes. For instance, `rep peer Object[]` is a subtype of `rep readonly Object[]` because the element type `peer Object` is a subtype of the element type `readonly Object`.

Like Java, the Universe type system allows downcasts of reference types. In



particular, it is possible to cast a readonly type into a peer or rep type. As with conventional downcasts, such casts need dynamic type checking to guarantee that the more specific ownership information of the subtype is accurate. Alternatively, static verification can be applied to show that the downcast is permitted. Consider a variable `roT` of type `readonly T`. The downcast `(rep T) roT` requires one to check the condition `roT == null || roT.owner == this`, that is, to check that `roT` actually contains a rep reference.

Type Rules

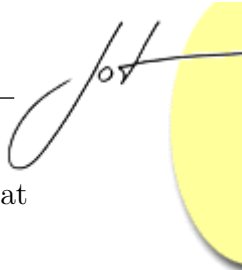
The type rules of the Universe type system guarantee that the ownership modifiers of reference types correctly reflect the owner of the referenced object. Moreover, they enforce the owner-as-modifier property. In this subsection, we present the most interesting type rules.

Assignment. The rule for an assignment is the standard Java rule: The type of the right-hand side expression has to be a subtype of the type of the left-hand side variable. This rule renders method `capture` of class `LinkedList` (Fig. 3) incorrect: The type of `n`, `peer Node`, is not a subtype of the type of `first`, `rep Node`.

Object Creation. An object creation expression is of the form `new peer T(...)` or `new rep T(...)`, where `T` is a class name. An array creation expression has two ownership modifiers. Analogous to object creation, the first ownership modifier has to be `peer` or `rep`. The element type can have a `peer` or `readonly` modifier. We forbid the `rep` modifier for element types because it is not possible to create objects owned by arrays. Such an object could be created only by methods executed on an array as receiver object, which is not possible in Java.

Field Access. To determine the owner of an object referenced by `x.f` — and, thus, the type of the field access `x.f` — one has to consider the ownership modifiers of both `x` and `f`:

1. If the types of both `x` and `f` are peer types, then we know (a) that the object referenced by `x` has the same owner as `this`, and (b) that the object referenced by `x.f` has the same owner as `x` and, thus, the same owner as `this`. Consequently, the type of `x.f` also has the modifier `peer`.
2. If the type of `f` is a rep type, then the type of `this.f` has the modifier `rep`, because the object referenced by `this.f` is owned by `this`.
3. If the type of `x` is a rep type and the type of `f` is a peer type, then the type of `x.f` has the modifier `rep`, because (a) the object referenced by `x` is owned



by `this`, and (b) the object referenced by $x.f$ has the same owner as x , that is, `this`.

4. In all other cases, we cannot determine statically that the object referenced by $x.f$ has the same owner as `this` or is owned by `this`. Therefore, in these cases the type of $x.f$ has the modifier `readonly`.

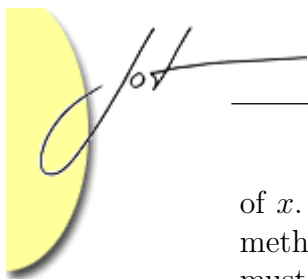
To enforce the owner-as-modifier property, we perform the following check, which corresponds to the assertion generated for field updates (see Sec. 2). If the field access $x.f$ occurs as left-hand side of an assignment, the type of x must not be a `readonly` type. This requirement is violated by the last assignment in method `exchangeFirst` (Fig. 3): The type of `l.first` is `readonly Node`. Therefore, an assignment to `l.first.elem`, which parses as `(l.first).elem`, is not allowed.

Method Call. Analogously to field accesses, the declared parameter and result types of a method have to be interpreted w.r.t. the type of the receiver expression of the method call. Consider a method with signature `void foo(peer T p)`. The `peer` modifier expresses that the parameter `p` has the same owner as the receiver object on which `foo` is executed. Therefore, if `foo` is called on an expression of a `rep` type, the actual parameter of the call must also be of a `rep` type to meet this requirement. This interpretation of parameter and result types is performed by the mapping described for field accesses, with the type of the field replaced by the parameter or result type of the method. In our example, the combination of a `rep` type (the type of the receiver) and a `peer` type (the type of `p`) yields a `rep` type (point 3 in the enumeration above).

For a call $x.m(\dots a_i \dots)$, this observation leads to the following conditions: (1) The type of an actual parameter a_i must be a subtype of the combination of the types of x and of the formal parameter p_i . (2) The type of the call expression is the combination of the type of x and the declared result type of m . (3) If at least one formal parameter of m has a `rep` type, then m may be called only on the receiver expression `this`. This restriction is necessary because if the receiver expression, x , is different from `this`, the combination of x 's type and a `rep` type yields a `readonly` type. In that case, callers of m could pass arguments in arbitrary contexts, although m expects an argument owned by its receiver, x . Rule (3) prevents this problem.

The owner-as-modifier property requires that `readonly` references must not be used to modify the referenced object. To prevent indirect modification through method calls, only side-effect free methods may be called on `readonly` references. We use JML's `pure` modifier for this check. This modifier indicates that a method is side-effect free. The JML compiler is supposed to check purity statically.

The `foo` example above illustrates a problem with pure methods: Assume that x is of a `readonly` type and that `foo` is pure. Since `foo`'s formal parameter `p` is of a `peer` type, a call $x.foo(y)$ requires that x and y have the same owner. However, this condition cannot be checked syntactically since x 's type does not specify the owner



of x . We solve this problem by an additional requirement: All parameters of pure methods (the only methods that can be called on expressions of readonly types) must have readonly types. This requirement does not restrict expressiveness since pure methods cannot modify their parameter objects anyway. Missing ownership modifiers for formal parameters of pure methods are defaulted to `readonly` by our implementation.

Universe Invariant

In the dynamic ownership encoding, `peer` and `rep` annotations are desugared into `requires` clauses, `ensures` clauses, and `invariants`. That is, they express ownership relations that hold in certain execution states, namely the pre- and poststates of method executions. The Universe type system makes a much stronger guarantee: The specified ownership relations hold in all execution states. For instance, method `bar` in Fig. 5 satisfies all assertions of the dynamic ownership encoding, but does not typecheck since the specified ownership relation is violated temporarily. Moreover, the Universe type system checks ownership properties of all expressions, whereas the dynamic encoding addresses only fields, method parameters, and method results.

```
public class Difference {
  /*@ rep @*/ Object o;
  public void bar() {
    o = new /*@ peer @*/ Object(); // breaks invariant temporarily
    o = new /*@ rep @*/ Object(); // reestablishes invariant
  }
}
```

Figure 5: A class illustrating the difference between checking ownership by assertions or by type checking. Method `bar` satisfies the assertions generated from the ownership annotations, but does not typecheck.

In summary, the Universe type system guarantees that the following program invariant holds in every execution state: If object X holds a direct reference to object Y then at least one of the following cases applies: (1) X and Y are in the same context, or (2) X is the owner of Y , or (3) the reference is `readonly`. W.r.t. this program invariant, local variables and formal parameters behave like instance variables of the `this` object.

4 COMBINING UNIVERSE TYPES AND JML SPECIFICATIONS

Readonly types can be used to leave the owner of an object X unspecified, which makes the Universe type system very flexible. Still, the owner of X and objects in



the same context as X can use downcasts to obtain a readwrite reference to X and modify X . JML annotations can be used to specify ownership relations that cannot be expressed in the Universe type system. In particular, these annotations can be used to prove that a downcast does not throw an exception if the annotations are satisfied.

In this section, we illustrate the flexibility of the combination of the Universe type system and JML ownership specifications by two examples: producer-consumer and doubly-linked lists with iterators. Both examples satisfy the owner-as-modifier property, but cannot be implemented in an owner-as-dominator model.

Producer-Consumer

Fig. 6 shows a producer and a consumer that share a common buffer. The producer puts products into the buffer, which are then retrieved by the consumer. The shared ring buffer is implemented as an array of `Product` objects, which is owned by the producer to protect it from unwanted modifications. The producer, the consumer, and the products are in the same context. The owner-as-modifier property allows the consumer to have a readonly reference to the buffer although it is owned by the producer. This reference is not permitted in owner-as-dominator models.

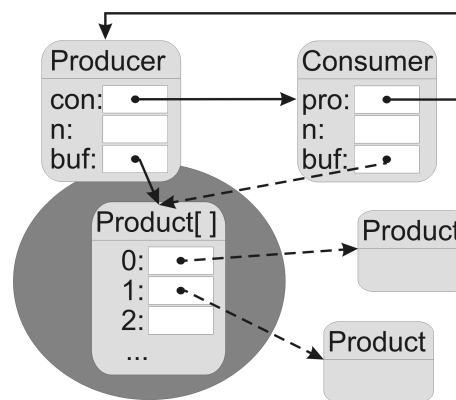


Figure 6: Object structure of the producer-consumer example. Producer, consumer, and product objects are in the same context. The context of objects owned by the producer is depicted by the ellipse. It contains the buffer array. Readwrite references are drawn with solid lines and readonly references with dashed lines.

We adopt Barnett and Naumann's implementation that synchronizes the producer and the consumer without requiring the consumer to modify the buffer [BN04b]. To achieve that, the producer and the consumer each store a buffer index and mutual peer references to each other. These references are used to relate the indices in specifications, in particular, to express that the buffer is empty or full.

```

public class Producer {
  /*@ spec_public rep readonly @*/ Product[] buf;
  /*@ spec_public @*/ int n;
  /*@ spec_public peer @*/ Consumer con;

  /*@ invariant buf != null && 0 <= n && n < buf.length &&
    @          (\forall int i; 0 <= i && i < buf.length;
    @          buf[i] == null || buf[i].owner == this.owner);
    @*/

  public Producer() {
    buf = new /*@ rep readonly @*/ Product[10];
  }

  /*@ requires con != null && con.n != n;
    @ ensures  n == \old((n+1) % buf.length);
    @*/
  public void produce(/*@ peer @*/ Product p) {
    buf[n] = p;
    n = (n+1) % buf.length;
  }
}

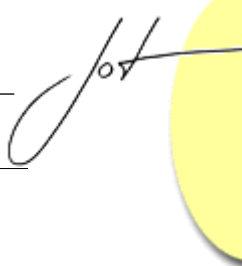
```

Figure 7: Implementation of the producer. The ownership information about the products stored in the buffer is expressed by an invariant.

The buffer stores readonly references to products. The invariant of class `Producer` (Fig. 7) specifies that the products in the buffer have the same owner as the producer. This invariant is established by `Producer`'s constructor because all array elements are initialized to `null`. The `produce` method maintains the invariant because the peer type of the formal parameter `p` guarantees that the inserted product has the same owner as the producer.

Since the buffer is owned by the producer, class `Consumer` (Fig. 8) has a readonly field for the reference to the buffer. The interface specification of the class expresses the synchronization conditions with the producer. This specification does not specify ownership relationships.

The most interesting aspect of class `Consumer` is method `consume`, which casts the readonly reference obtained from the buffer into a peer reference. This downcast is guarded by the runtime check `buf[n] == null || buf[n].owner == this.owner`. This check cannot cause a runtime error, because (1) by `Producer`'s invariant, we have `pro.buf[n] == null || pro.buf[n].owner == pro.owner`; (2) by `Consumer`'s invariant, we have `buf == pro.buf`; (3) by the type of field `pro` and `Consumer`'s invariant, we have `pro.owner == this.owner`. This example



```

public class Consumer {
  /*@ spec_public readonly readonly @*/ Product[] buf;
  /*@ spec_public @*/ int n;
  /*@ spec_public peer @*/ Producer pro;

  /*@ invariant buf != null && 0 <= n && n < buf.length;
  /*@ invariant pro != null && pro.con == this && pro.buf == buf;

  /*@ requires p != null && p.con == null;
  public Consumer(/*@ peer @*/ Producer p) {
    buf = p.buf;
    pro = p;
    n = buf.length-1;
    p.con = this;
  }

  /*@ requires (n+1) % buf.length < pro.n;
  @ ensures n == \old((n+1) % buf.length);
  @*/
  public /*@ peer @*/ Product consume() {
    n = (n+1) % buf.length;
    return (/*@ peer @*/ Product) buf[n];
  }
}

```

Figure 8: Implementation of the consumer. Method `consume` casts a readonly reference into a peer reference. The invariant of `Producer` guarantees that this cast does not lead to a runtime error.

shows how Universe type information and JML specifications can be combined to reason about ownership properties.

Collections with Iterators

Iterators typically have direct references to the internal representation of the collection they iterate on. Such a reference can be used to read and modify the collection. In this example, we show how to implement modifying iterators in the owner-as-modifier model. Moreover, we illustrate that readonly references can be used to implement methods that traverse object structures, for instance, to perform a deep comparison or to clone the structure.

Fig. 9 shows the object structure of a doubly-linked list with one iterator. Like in class `Consumer`, the iterator uses a readonly reference to directly access the objects

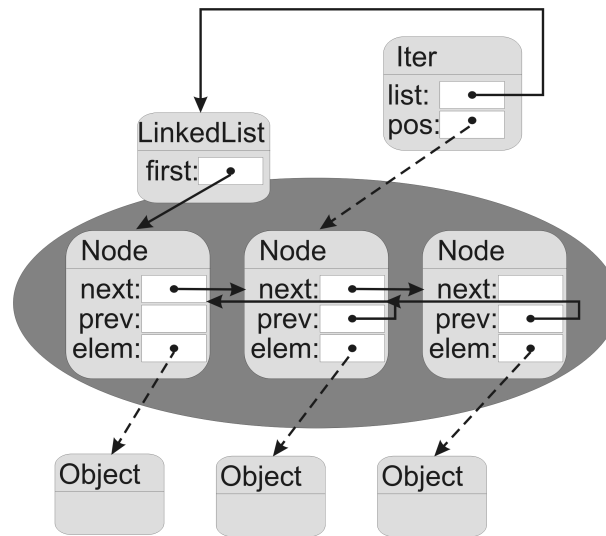


Figure 9: Object structure of the collection example. The `LinkedList` object owns the nodes of the doubly-linked list. The iterator is in the same context as the list head. It has a reference to the list head and a readonly reference to the `Node` object at the iterator position.

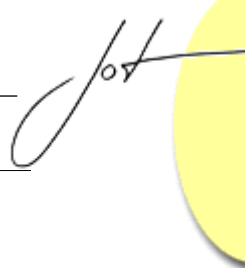
owned by another object, in this case the `Node` objects owned by the list head. Since the elements stored in the list are referenced readonly, they can be owned by any object.

The implementations of the head (class `LinkedList`), the nodes (class `Node`), and the iterators (class `Iter`) of the doubly-linked list are shown in Figs. 10, 4, and 11, resp.

Modifying Iterators. The owner-as-modifier model allows iterators to have readonly references to the nodes of the list. Modifications of the list have to be initiated by methods of the list head. Iterators can modify the list structure indirectly by delegating method calls to the head.

Class `LinkedList` offers a non-public method `set` that can be called by iterators to store an object in a given node. `set` takes a node of the list as parameter. This parameter has to be provided by the calling iterator. Since iterators can only have readonly references to list nodes, the type of the parameter, `np`, is `readonly Node`. Before updating `np.elem`, method `set` performs a downcast to obtain a writable reference to the `Node` object. This downcast requires `np` to be owned by the list head, which is expressed by the `requires` clause of `set`.

Method `set` illustrates how Universe and JML annotations complement each other: Although parameter `np` has to be owned by `this`, it cannot be declared with a `rep` type because methods with `rep` parameters may be called only on the receiver



```

public class LinkedList {
  /*@ spec_public rep @*/ Node first;

  /*@ requires np != null && np.owner == this;
  void set(/*@ readonly @*/ Node np, /*@ readonly @*/ Object e) {
    /*@ rep @*/ Node n = (/*@ rep @*/ Node) np;
    n.elem = e;
  }

  public /*@ pure @*/ boolean equals(/*@ readonly @*/ Object l) {
    if (!(l instanceof /*@ readonly @*/ LinkedList))
      return false;
    /*@ readonly @*/ Node f1 = first;
    /*@ readonly @*/ Node f2 = ((/*@ readonly @*/ LinkedList)l).first;
    while (f1 != null && f2 != null && f1.elem == f2.elem) {
      f1 = f1.next;
      f2 = f2.next;
    }
    return f1 == null && f2 == null;
  }
  // constructors and other methods omitted
}

```

Figure 10: Implementation of the list head. The method `set` allows iterators to modify the value stored in a `Node` object.

`this` (see rule (3) for methods calls, Sec. 3). In particular, iterators would not be allowed to call method `set`. To solve this problem, we use a `readonly` type for `np` and express the necessary stronger ownership information by a `requires` clause.

Each iterator is associated with a list. Objects of class `Iter` store a reference to the list they belong to. Field `pos` contains a `readonly` reference to the node at the current iterator position. The invariant expresses that the referenced node is owned by the list, a property, that cannot be expressed by ownership type systems. Since `first` is of a `rep` type, this invariant is established by `Iter`'s constructor.

Although an iterator does not own the `Node` object it points to, it can modify its state by `Iter`'s `setValue` method. This method delegates the call to `LinkedList.set`. The invariant of class `Iter` guarantees that the `requires` clause of `set` is satisfied.

According to Noble's classification [Nob00], this iterator design is a blend between a structure-sharing external iterator and an external iterator: Read operations are performed directly on the list's representation, whereas write operations access the representation via the list interface.

```

public class Iter {
  /*@ spec_public peer @*/    LinkedList list;
  /*@ spec_public readonly @*/ Node    pos;

  /*@ invariant pos != null && pos.owner == list;

  public Iter(/*@ peer @*/ LinkedList l) {
    list = l;
    pos = l.first;
  }

  public void setValue(/*@ readonly @*/ Object e) {
    list.set(pos, e);
  }
  // other methods omitted
}

```

Figure 11: Implementation of the list iterator. The peer reference to the list head is used to modify the list.

Traversing Object Structures. `LinkedList`'s `equals` performs a deep comparison of two lists. This method traverses the nodes owned by `this` and the parameter `l`. In the owner-as-modifier model, the nodes of `l` can be accessed through readonly references.

The `instanceof` and `cast` expressions in method `equals` are necessary to make sure the parameter `l` is a `LinkedList`. Since `peer` and `rep` types are subtypes of the corresponding readonly types, `instanceof` and `cast` expressions for readonly types check the standard Java type, but no ownership information. `Peer` and `rep` types can be used in `instanceof` and `cast` expressions to check in addition that an object has the same owner as `this` or is owned by `this`, resp.

5 IMPLEMENTATION

We integrated the Universe type system into the MultiJava compiler [CMLC04] on which the JML compiler is built. In this section, we describe the JML implementation without distinguishing whether a feature is actually implemented in MultiJava or JML.

Our implementation covers the type system described in Sec. 3. The type checker is applied to both Java implementations and JML specifications. Typechecking specifications, for instance, requires that the `owner` field is declared with a `readonly` modifier:



```
/*@ ghost public readonly Object owner; @*/
```

Alternatively, one could implicitly treat all references in specifications as `readonly` since the evaluation of a specification must not modify existing objects anyway. We use the Universe type checking for specifications to simplify the implementation.

Defaulting and Backwards Compatibility

Our implementation reduces the annotation overhead significantly by a simple defaulting scheme. The default ownership modifier for reference types is `peer`. Defaulting reference types to `peer` types allows most Java programs without ownership annotations to be typechecked by the Universe type system: all objects are in the root context and can reference and modify each other.

We depart from the `peer` default in two cases. (1) Since parameter types of pure methods must be `readonly` (see Sec. 3), we use `readonly` as default for these types. (2) The exception types in `throws` and `catch` clauses are defaulted to `readonly`, which allows exceptions to be propagated to handlers in arbitrary contexts (see our earlier work [DM04] for details).

Standard Java programs are not accepted by the Universe type checker if they expect a reference that is by default `readonly` to be `readwrite`, for instance, if they modify the state of a caught exception object. These uncommon cases can be fixed easily, for instance by inserting a downcast to a `peer` type or by creating and re-throwing a new exception object. With the concrete syntax of ownership modifiers used so far, some standard Java programs cannot be typechecked with the Universe type system because they use the keywords `peer`, `rep`, or `readonly` as identifiers.

To be able to use the JML compiler for *all* Java programs, the Universe type checking can be controlled in a fine-grained way by command line switches. Users can choose between four modes of operation: (1) the Universe type system is switched off completely; (2) the ownership annotations are parsed, but typechecking is switched off; (3) Universe type checking is switched on; (4) Universe type checking is switched on and the necessary runtime checks are generated.

To avoid syntactic conflicts with the keywords `peer`, `rep`, and `readonly`, programmers can use an alternative syntax for ownership modifiers, where the keywords are preceded by a backslash, for example `/*@ \peer @*/`. Such modifiers are ignored by the compiler if the Universe type system is switched off (modes 1 and 2). This version of the modifiers can be used for Java API classes that should be usable either with or without enabled Universe type system. Finally, it is possible to use `peer`, `rep`, `readonly`, and `pure` without enclosing comments. However, programs with this concise syntax cannot be compiled by standard Java compilers.



Runtime Support

In this subsection, we describe how the runtime checks of the Universe type system are implemented. Based on this implementation, the checks for the dynamic ownership encoding (Sec. 2) can be added easily.

Like standard Java, the Universe type system requires runtime checks for downcasts and array updates. Besides the plain Java types, these checks have to compare the ownership information. Ownership information at runtime is also necessary to evaluate `instanceof` expressions.

Representation of Ownership Information. The necessary runtime checks can be expressed as assertions in terms of the ghost field `owner`. The JML runtime assertion checker handles ghost fields by adding a normal field to the declaring class and appropriate get and set methods. However, this approach works only for ghost fields of classes that are compiled by the JML compiler, which is not the case for `Object`. Neither modifying `Object`'s source code nor patching its class file is an option since the distribution of a modified version of `Object` violates the Sun license terms.

We solve this problem by storing ownership information externally in a global hashtable. This table maps objects to their owner object. For array objects, we also store the ownership modifier of the element type in the hashtable. This information is used in the runtime checks of array updates. We use Java's weak references to ensure that storing a reference in the hashtable does not affect garbage collection (see [Sch04] for details).

As described in Sec. 2, conceptually the `owner` field is set by `Object`'s default constructor. Since we cannot modify the implementation of `Object`, we add a new object to the ownership hashtable at two places: (1) before the first statement of the constructor, which ensures that the ownership information of the new object is available during the execution of the constructor; (2) after the `new` expression, which ensures that the new object is added even if the constructor was not compiled by the JML compiler. With this solution, it is still possible to create objects that are not added to the ownership hashtable if the `new` expression occurs in a class that is not compiled by the JML compiler. A Java system property is used to control whether runtime checks for such objects always pass or always fail.

Runtime Checks. We implemented the runtime checks for downcasts, array updates, and `instanceof` expressions as additional bytecode instructions generated by the compiler. As future work, we plan to adapt JML's runtime assertion checker to map accesses to the `owner` ghost field to accesses to the ownership hashtable.

For downcasts and array updates, the result of a failed check can be controlled by Java system properties. The options range from throwing an exception to only reporting the error on the console.



The checks for downcasts and `instanceof` are comparisons of the corresponding entries in the ownership hashtable. Like Java, the Universe type system has covariant array subtyping (see Sec. 3). For instance, `rep peer Object[]` is a subtype of `rep readonly Object[]`. Therefore, an array variable with a static readonly element type could, at runtime, contain an array with peer element type. Consequently, updating such an array requires a runtime check that the reference assigned to the array element actually is a peer reference. The ownership hashtable stores the element ownership modifier of each array. This modifier is used to check whether the owner of the object on the right-hand side of the update conforms to the element type of the array.

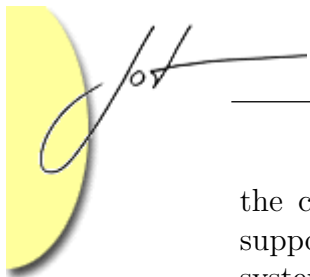
6 RELATED WORK

Clarke *et al.* [CPN98, Cla01] developed the first of a number of ownership type systems that enforce the owner-as-dominator property. In addition to the readwrite references permitted by the Universe type system, Clarke *et al.*'s work allows an object, X , to reference objects in ancestor contexts of the context that contains X . Such references violate neither the owner-as-dominator nor the owner-as-modifier property. Still, we require references to ancestor contexts to be readonly to prevent methods from modifying objects in ancestor contexts because such modifications are difficult to handle by state-of-the-art specification techniques for frame properties [MPHL03]. Clarke *et al.* use context parameters to express role separation. Readonly types can replace context parameters in many situations, impose less annotation overhead, and lead to programs that are easier to read and reason about. Furthermore, readonly references allow multiple objects to reference one representation, which is not supported by the owner-as-dominator model used in ownership types. However, such non-owning references to a representation are used in common implementations such as iterators or shared data structures.

Clarke and Drossopoulou [CD02] extended the original ownership type system to support inheritance. Their type system is ownership parametric and enforces the owner-as-dominator property. Therefore, it suffers from the same problems as the original ownership type system. Based on their type system, Clarke and Drossopoulou present an effects system and use it to reason about aliasing and non-interference.

In recent work, Noble *et al.* [PNCB03, PNCR04] combined ownership and type parameterization by introducing generic ownership. This proposal reduces the annotation overhead for generic classes, but does not address the other shortcomings of ownership types.

SafeJava [Boy04, BLS03] is very similar to ownership types, but supports a model that is slightly less restrictive than owner-as-dominator: An object and all associated instances of inner classes can access a common representation. For instance, iterators can be implemented as inner class of the collection and, therefore, directly reference



the collection's representation. However, more general forms of sharing are not supported. SafeJava is more flexible than ownership types, but the Universe type system is both syntactically simpler and more expressive. SafeJava has been applied to prevent data races and deadlocks.

Boyapati *et al.* [BLR03] present a space-efficient implementation of downcasts in SafeJava. Their implementation inspects each class, C , to determine whether downcasts for C objects potentially require dynamic ownership information. If not, ownership information is not stored for C objects. This optimization does not work for the Universe type system, where readonly references to objects of any class can be cast to peer or rep references and, therefore, objects of every class potentially need runtime ownership information.

Ownership domains [AC04] support a model that is less restrictive than owner-as-dominator. Contexts can be structured into several domains. Domains can be declared public, which permits reference chains to objects in the public domain that do not pass through their owner. Programmers can control whether objects in different domains can reference each other. For instance, iterators in a public domain of a collection are accessible for clients of the collection. They can be allowed to reference the representation of the collection stored in another domain. The concept of ownership domains is powerful and allows many forms of sharing. However, its suitability to support verification of functional correctness properties is unclear. For instance, placing the buffer of the producer-consumer example in the producer's public domain allows the consumer as well as all other objects in the same context to modify the buffer. This makes it difficult, if not impossible, to maintain invariants on the buffer. Supporting verification has been the main motivation behind the Universe type system. Another drawback of ownership domains is the annotation overhead they impose. Like ownership types, ownership domains impose the annotation overhead of context parameters.

Confined types [BV99] guarantee that objects of a confined type cannot be referenced in code declared outside the confining package. Confined types have been designed for the development of secure systems. They do not support representation encapsulation on the object level.

Several attempts at inferring ownership types [AKC02, AS04, Wre03] showed that the complexity of parametric ownership type systems makes inference difficult. We are working on an inference system for Universe types and expect that the simplicity of our type system eases inference. The experiences so far are promising.

The Boogie methodology [LM04] for reasoning about invariants is based on a dynamic ownership encoding similar to the one described in this paper. In this methodology, any reference can be used to modify an object, provided that all transitive owners of this object are made mutable by applying a special unpack operation. In practice, this requirement is typically met by following the owner-as-modifier policy: the owner unpacks itself before initiating the modification of an owned object. The Boogie methodology supports ownership transfer.



Banerjee and Naumann use ownership to prove a representation independence result for object-oriented programs [BN04a]. Their ownership model requires that for a given pair of classes C, D , all instances of D are owned by some instance of C . This is clearly too restrictive for many implementations. For instance, lists are typically used as internal representation by many classes. Similarly, it is unclear how arrays can be supported by such a model. Banerjee and Naumann present a static analysis to check whether a program satisfies the ownership model for a pair of classes C, D .

Unique references and linear types [Boy01, BNR01, FD02, Wad90] can be used for a very restrictive form of alias control. For ownership type systems, a weaker form of uniqueness [CW03] is sufficient to enable ownership transfer.

Skoglund [Sko02] as well as Birka and Ernst [BE04] present type systems for readonly types that are similar to ours. Birka and Ernst's type system is more flexible than ours as it allows one to exclude certain fields or objects from the immutable state. Neither Skoglund nor Birka and Ernst combined readonly types with ownership.

Our readonly types leave the owner of an object unspecified. Whenever precise information about the owner is needed, a downcast with a dynamic type check is used. This approach is similar to soft typing [CF91], where a compiler does not reject programs that contain potential type errors, but rather introduces runtime checks around suspect statements. In soft typing, these runtime checks are added automatically by the compiler whereas we require programmers to introduce casts manually.

7 CONCLUSION

We have shown that the combination of a lightweight ownership type system and JML ownership specifications can handle interesting implementations while keeping the annotation and checking effort small. Our implementation of the Universe type system in the JML compiler allows one to apply ownership-based verification techniques to programs specified in JML.

We are currently using our implementation in an industrial case study, which investigates the practicality of the approach described in this paper. The intermediate results are promising: Most classes can be handled by the Universe type system, sometimes by making minor modifications to the code.

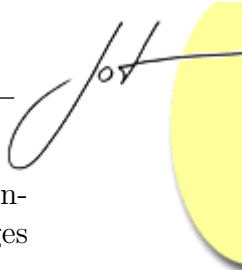
As future work, we plan to extend the Java subset supported by the Universe type system. In particular, we want to handle static fields, generics, and reflection. Other threads of future work are the development of an inference tool for the Universe type system and the integration of the type system in the verification tools ESC/Java2 and Jive.

ACKNOWLEDGMENTS

We are grateful to Gary Leavens for many helpful discussions about the integration of the Universe type system in to JML. The reviewers made valuable suggestions that improved the paper. We also thank Daniel Schrengenberger for his help with the implementation of the runtime checks.

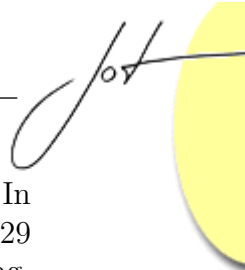
REFERENCES

- [AC04] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 1–25. Springer-Verlag, 2004.
- [AKC02] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2002.
- [AS04] R. Agarwal and S. D. Stoller. Type inference for parameterized race-free Java. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 2937 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.
- [BCC⁺03] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Formal Methods for Industrial Critical Systems (FMICS)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, 2003.
- [BDF⁺04] 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), 2004. www.jot.fm.
- [BE04] A. Birka and M. Ernst. A practical type system and language for reference immutability. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2004.
- [BLR02] 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.
- [BLR03] C. Boyapati, R. Lee, and M. Rinard. Safe runtime downcasts with ownership types. In *ECOOP International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*, 2003.

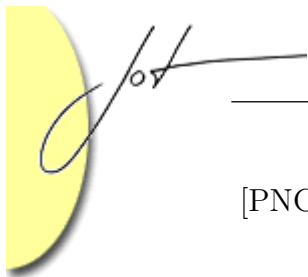


- [BLS03] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Principles of Programming Languages (POPL)*, pages 213–223. ACM Press, 2003.
- [BN02] A. Banerjee and D. A. Naumann. Representation independence, confinement, and access control. In *Principles of Programming Languages (POPL)*, pages 166–177. ACM Press, 2002.
- [BN04a] A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. Technical Report 2004-14, Stevens Institute of Technology, 2004.
- [BN04b] M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *Mathematics of Program Construction (MPC)*, Lecture Notes in Computer Science. Springer-Verlag, 2004.
- [BNR01] J. Boyland, J. Noble, and W. Retert. Capabilities for aliasing: A generalisation of uniqueness and read-only. In J. Lindskov Knudsen, editor, *Object-Oriented Programming (ECOOP)*, number 2072 in Lecture Notes in Computer Science, pages 2–27. Springer-Verlag, 2001.
- [Boy01] J. Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, 2001.
- [Boy04] C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT, 2004.
- [BRL03] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Formal Methods (FME)*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer-Verlag, 2003.
- [BV99] B. Bokowski and J. Vitek. Confined types. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices, 1999.
- [CD02] 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.
- [CF91] R. Cartwright and M. Fagan. Soft typing. *SIGPLAN*, 26(6):278–292, 1991. Programming Language Design and Implementation (PLDI).
- [Che03] Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Iowa State University, 2003.

- [CK04] D. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS)*, volume 3362 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.
- [CL02] Y. Cheon and G. T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In H. R. Arabnia and Y. Mun, editors, *Software Engineering Research and Practice (SERP)*, pages 322–328. CSREA Press, 2002.
- [Cla01] D. Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, 2001.
- [CMLC04] C. Clifton, T. Millstein, G. T. Leavens, and Chambers C. MultiJava: Design rationale, compiler implementation, and applications. Technical Report 04-01b, Iowa State University, Dept. of Computer Science, 2004. Accepted for publication, pending revision.
- [CPN98] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, 1998.
- [CW03] D. G. Clarke and T. Wrigstad. External uniqueness is unique enough. In L. Cardelli, editor, *European Conference for Object-Oriented Programming (ECOOP)*, volume 2743 of *Lecture Notes in Computer Science*, pages 176–200. Springer-Verlag, 2003.
- [DM04] W. Dietl and P. Müller. Exceptions in ownership type systems. In E. Poll, editor, *Formal Techniques for Java-like Programs*, pages 49–54, 2004.
- [FD02] M. Fähndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *Programming Language Design and Implementation (PLDI)*, pages 13–24. ACM Press, 2002.
- [FLL⁺02] 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)*, pages 234–245. ACM Press, 2002.
- [Jac04] B. Jacobs. Weakest precondition reasoning for Java programs with JML annotations. *Journal of Logic and Algebraic Programming*, 58:61–88, 2004.



- [JP01] B. Jacobs and E. Poll. A logic for the Java modeling language JML. In *Fundamental Approaches to Software Engineering (FASE)*, volume 2029 of *Lecture Notes in Computer Science*, pages 284–299. Springer-Verlag, 2001.
- [LBR99] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.
- [LBR04] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev27, Iowa State University, Department of Computer Science, 2004. See www.jmlspecs.org.
- [LM04] 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 *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.
- [MPH01] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.
- [MPHL03] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15:117–154, 2003.
- [MPHL04] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. Technical Report 424, Department of Computer Science, ETH Zurich, 2004.
- [Mül02] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
- [Nob00] J. Noble. Iterators and encapsulation. In *TOOLS '00: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)*, page 431. IEEE Computer Society, 2000.
- [NVP98] J. Noble, J. Vitek, and J. M. Potter. Flexible alias protection. In E. Jul, editor, *ECOOP '98: Object-Oriented Programming*, volume 1445 of *LNCS*. Springer-Verlag, 1998.
- [PNCB03] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership. Technical Report CS-TR-03-16, Victoria University of Wellington, 2003.



- [PNCR04] A. Potanin, J. Noble, D. Clarke, and Biddle R. Featherweight generic confinement. In *Foundations of Object-Oriented Languages (FOOL)*, 2004.
- [Sch04] D. Schrengenberger. Runtime checks for the Universe type system. Semester project, available from http://sct.inf.ethz.ch/projects/student_docs/Daniel_Schrengenberger/Daniel_Schrengenberger_SA_paper.pdf, 2004.
- [Sko02] M. Skoglund. Sharing objects by read-only references. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology (AMAST)*, volume 2422 of *Lecture Notes in Computer Science*, pages 457–472. Springer-Verlag, 2002.
- [Wad90] P. Wadler. Linear types can change the world! In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods (PROCOMET)*, 1990.
- [Wre03] A. Wren. Inferring ownership. Master’s thesis, Department of Computing, Imperial College, June 2003.

ABOUT THE AUTHORS



Werner Dietl is a PhD student and research assistant at ETH Zurich. He works on the combination of ownership type systems and software verification. Email: werner.dietl@inf.ethz.ch.



Peter Müller is assistant professor and head of the Software Component Technology Group at ETH Zurich. His research focuses on specification and verification of object-oriented software. Email: peter.mueller@inf.ethz.ch.