Verification of Multithreaded Object-Oriented Programs with Invariants

Bart Jacobs[*]
Dept. of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium
bart.jacobs@cs.kuleuven.ac.be

K. Rustan M. Leino
Microsoft Research
One Microsoft Way
Redmond, WA, USA
leino@microsoft.com

Wolfram Schulte
Microsoft Research
One Microsoft Way
Redmond, WA, USA
schulte@microsoft.com

ABSTRACT

Developing safe multithreaded software systems is difficult due to the potential unwanted interference among concurrent threads. This paper presents a sound, modular, and simple verification technique for multithreaded object-oriented programs with object invariants. Based on a recent methodology for object invariants in single-threaded programs, this new verification technique enables leak-proof ownership domains. These domains guarantee that only one thread at a time can access a confined object.

0. INTRODUCTION

A primary aim of a reliable software system is ensuring that all objects in the system maintain *consistent* states: states in which all fields, and all fields of other objects on which they depend, contain legal meaningful values. In this paper, we formalize consistency constraints as *object invariants*, which are predicates over fields.

An object is consistent if it is in a state where its invariant must hold. We also allow an object to be in a mutable state, where its invariant may temporarily be violated.

It is hard to maintain object invariants in sequential programs, and it is even harder in concurrent programs. For example, consider the following method:

$$\begin{array}{l} \textbf{void}\ \mathit{Transfer}(\mathit{DualAccounts}\ o, \textbf{int}\ \mathit{amount})\ \{ \\ \quad o.a := o.a - \mathit{amount}\ ; \\ \quad o.b := o.b + \mathit{amount}\ ; \\ \} \end{array}$$

Suppose this method is to maintain the invariant that for all dual accounts d: $d.a + d.b = 0$. In a concurrent setting, this invariant can be violated in several ways. Even if the programming system ensures that each read or write of a field is atomic, the interleavings might cause the invariant to be violated. For example, if one thread executes method *Transfer* and reads $o.a$, but before the thread performs the write to $o.a$, another thread causes some update of $o.a$, then the invariant will not be maintained.

In a concurrent setting, consistency of an object can be ensured by exclusion at a level coarser than individual reads and writes. For example, while one thread updates an object, another is not allowed to perform any operation on the object. In contemporary object-oriented languages, exclusion is implemented via locking.

Guaranteed exclusion simplifies the automatic verification of multithreaded code a lot. It means that we can simply split the proof of the concurrent program into a proof for exclusion and a proof for a sequential program [17].

In this paper, we present a new programming methodology for sound modular verification of multithreaded object-oriented programs with object invariants. The methodology not only guarantees that every object protects itself from consistency violations, but it also allows aggregates of objects to define *leak-proof ownership domains*. These domains guarantee that only one thread at a time can access an object of the aggregate.

The methodology achieves modular static verification by requiring methods to be annotated with simple ownership requirements. The methodology is a extension of the Boogie methodology for sequential code, as described in our previous work [1].

The paper proceeds as follows. The next three sections gradually introduce our methodology: Section 1 introduces object invariants, Section 2 introduces confinement within objects, and Section 3 presents our extension to confinement within threads. In Section 4, we sketch a proof of the soundness of our verification method. We discuss additional issues of static verification in Section 5, of implementation in Java and C# in Section 6, and of run-time checking in Section 7. Sections 8 and 9 mention related work and conclude.

1. OBJECT INVARIANTS

We consider an object-oriented programming language with classes, for example like the class in Figure 0. Each class can declare an invariant, which is a predicate on the fields of an object of the class.

To allow a program temporarily to violate an object's invariant, the Boogie methodology [1] introduces into each object an auxiliary boolean field called inv.[0] We say that an object o is *consistent* if $o.inv = true$, otherwise we say the object is *mutable*. Only in the mutable state is the object's invariant allowed to be violated. The inv field can be mentioned in method contracts (*i.e.*, pre- and postconditions). It cannot be mentioned in invariants or in program

[*]Bart Jacobs co-authored this paper during an internship at Microsoft Research. Bart Jacobs is a Research Assistant of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.-Vlaanderen).

[0]The Boogie methodology also deals with subclasses, but for brevity we here consider only classes without inheritance. Extending what we say to subclasses is straightforward.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAVCBS 2004 Newport Beach, CA, USA

```
class IntList {
  rep int[] elems := new int[10] ;
  int count := 0 ;
  invariant 0 ≤ count ∧ count ≤ elems.Length ;

  void Add(int elem)
    requires inv ;
  {
    unpack (this) ;
    if (count = elems.Length)
      { elems := elems.Copy(count * 2) ; }
    elems[count] := elem ;
    count := count + 1 ;
    pack (this) ;
  }
}
```

Figure 0: An example class, representing a extensible list of integers. The invariant links the $count$ field with the array length of the $elems$ field. The Add method maintains the invariant.

code. The inv field can be changed only by two special statements, **unpack** and **pack**. These statements delineate the scope in which an object is allowed to enter a state where its invariant does not hold.

The rules for maintaining object invariants are as follows:

- A new object is initially mutable.
- Packing an object takes it from a mutable state to a consistent state, provided its invariant holds.
- Unpacking an object takes it from a consistent state to a mutable state.
- A field assignment is allowed only if the target object is mutable.

We formalize these rules as follows, where $Inv_T(o)$ stands for the invariant of class T applied to instance o.

```
pack_T o  ≡
  assert o ≠ null ∧ ¬o.inv ∧ Inv_T(o) ;
  o.inv ← true

unpack_T o  ≡
  assert o ≠ null ∧ o.inv ;
  o.inv ← false

o.f := E  ≡
  assert o ≠ null ∧ ¬o.inv ;
  o.f ← E
```

In this formalization, an **assert** statement checks the given condition and aborts program execution if the condition does not hold.

Our methodology guarantees the following program invariant for all reachable states, for each class T:

PROGRAM INVARIANT 0.

$$(\forall o\colon T \bullet\ o.inv \implies Inv_T(o))$$

Here and throughout, quantifications are over non-null allocated objects.

```
class Account {
  rep IntList hist := new IntList() ;
  int bal := 0 ;
  invariant bal =
      ( Σ i | 0 ≤ i < hist.count • hist.elems[i] ) ;

  void Deposit(int amount)
    requires inv ;
    ensures bal = old(bal) + amount ;
  {
    unpack (this) ;
    hist.Add(amount) ;
    bal := bal + amount ;
    pack (this) ;
  }
}
```

Figure 1: An example class illustrating aggregate objects.

2. CONFINEMENT WITHIN OBJECTS

The accessibility modifiers (like **private** and **public**) in contemporary object-oriented languages cannot guarantee consistency. Consider for example the class $Account$ in Figure 1, which uses an $IntList$ object to represent the history of all deposits ever made to a bank account. A bank account also holds the current balance, which is the same as the sum of the history, as is captured by the invariant.

We say an $Account$ object is an *aggregate*: its *part* is the object referenced through the field $hist$. Part objects are also known as *representation objects*. We qualify fields holding representation objects with a **rep** modifier (*cf.* [16]).

A part is said to be *leaked* if it is accessible outside the aggregate. In a sequential setting, leaking is not considered harmful, as long as the parts are leaked only for reading [15, 1].

An aggregate *owns* its parts. Object ownership, here technically defined via **rep** fields, establishes a hierarchy among objects. Invariants and ownership are related as follows: the invariant of an object o can depend only on the fields of o and on the fields of objects reachable from o by dereferencing only **rep** fields. (We don't allow an invariant to mention any quantification over objects.)

To formulate ownership properly, we introduce for each object an $owner$ field. Like inv, the $owner$ field cannot be mentioned in program code. We say an object o is *free* if $o.owner = null$. An object is *sealed* if it has a non-null owner object and that owner is consistent. The *ownership domain* of an object o is the set collecting o and all objects that o transitively owns. The rules for **pack** and **unpack** enforce that ownership domains are packed and unpacked only according to their order in the ownership hierarchy. Furthermore, **pack** and **unpack** change the ownership of representation objects as described by the following rules, which extend the ones given earlier.[1] We use the function $RepFields_T$ to denote the fields marked **rep** within class T.

```
pack_T o  ≡
  assert o ≠ null ∧ ¬o.inv ∧ Inv_T(o) ;
  foreach (f ∈ RepFields_T  where o.f ≠ null)
    { assert o.f.inv ∧ o.f.owner = null ; }
  foreach (f ∈ RepFields_T  where o.f ≠ null)
    { o.f.owner ← o ; }
  o.inv ← true
```

[1]This is a slightly different use of the $owner$ field than in [13].

$\mathbf{unpack}_T\ o \ \equiv$
  assert $o \neq \mathit{null} \wedge o.\mathit{inv}$;
  $o.\mathit{inv} \leftarrow \mathit{false}$;
  foreach $(f \in \mathit{RepFields}_T$ **where** $o.f \neq \mathit{null})$
    $\{\ o.f.\mathit{owner} \leftarrow \mathit{null}\ ;\ \}$

For illustration purposes, let us inspect a trace of the invocation $\mathit{acct}.\mathit{Deposit}(100)$ for a non-null $\mathit{Account}$ object acct that satisfies the precondition of $\mathit{Deposit}$, where we focus only on the involved inv and owner fields of the involved objects. First, $\mathit{Deposit}$ unpacks acct: acct is made mutable, hist is made free. Next, Add is called, which first unpacks hist and makes it mutable. Next, the updates happen. On return from the Add method, hist is packed again: the invariant of hist is checked and hist is made consistent. Finally, the $\mathit{Deposit}$ method packs acct: the invariant of acct is checked, acct is made consistent, and hist is sealed. And that's exactly our pre-state restricted to inv and owner fields of the objects in the ownership domain.

Generalizing from this example, we observe that the methodology ensures the following program invariant, for each class T:

PROGRAM INVARIANT 1.

$$(\forall\, o\colon T \bullet\ o.\mathit{inv} \implies \mathit{Inv}_T(o)\ ) \wedge$$
$$(\forall\, f \in \mathit{RepFields}_T, o\colon T \bullet\ o.\mathit{inv} \implies o.f = \mathit{null} \vee o.f.\mathit{owner} = o\ ) \wedge$$
$$(\forall\, o\colon T \bullet\ o.\mathit{owner} \neq \mathit{null} \implies o.\mathit{inv}\ )$$

3. CONFINEMENT WITHIN THREADS

In the object ownership scheme above, objects are either part of an aggregate object or they are free, which means they do not have any owner. For modular verification of multithreaded code, we now refine this scheme again. We say that an object can either be *free*, it can be *owned by an aggregate object*, or it can be *owned by a thread*. Correspondingly, the owner field is null, an object, or a thread.[2]

To support sequential reasoning about field accesses, we require a thread to have exclusive access to the fields during the execution of the program fragment to which the sequential reasoning applies. We require a thread to transitively own an object whenever it reads one of its fields, and to directly own an object whenever it writes one of its fields. Since no two threads can (transitively) own the same object concurrently, this guarantees exclusion.

The rules for thread ownership are as follows:

- A thread owns any object that it creates, and the new object is initially mutable.
- A thread can additionally attempt to **acquire** any object. This operation will block until the object is free. At that point, we know that the object is consistent and the thread gains ownership of the object.
- A thread can relinquish ownership of a consistent object using the **release** statement.
- A thread that owns a consistent aggregate object can gain ownership of its sealed representation objects by unpacking the aggregate object using the **unpack** statement. This transfers ownership of the representation objects from the aggregate object to the thread.
- A thread can, via a **pack** statement, transfer ownership of a consistent object that it owns to an aggregate object.
- A thread can perform a field assignment only if it owns the target object and the target object is mutable.
- A thread can read a field only if it transitively owns the target object. We actually enforce this rule by a slightly stricter rule: a thread can evaluate an access expression $o.f_1.\cdots.f_n.g$ only if it owns o and each object in the sequence $o.f_1.\cdots.f_n$ owns the next one.

These rules are an extension of the rules presented in the previous section. They give rise to the object lifecycle shown in Figure 2. Fully spelled out, they are formalized as follows, where we denote the currently executing thread by **tid** .

$\mathbf{pack}_T\ o \ \equiv$
  assert $o \neq \mathit{null} \wedge o.\mathit{owner} = \mathbf{tid} \wedge \neg o.\mathit{inv}$;
  foreach $(f \in \mathit{RepFields}_T$ **where** $o.f \neq \mathit{null})$
    $\{$ **assert** $o.f.\mathit{owner} = \mathbf{tid} \wedge o.f.\mathit{inv}\ ;\ \}$
  foreach $(f \in \mathit{RepFields}_T$ **where** $o.f \neq \mathit{null})$
    $\{\ o.f.\mathit{owner} \leftarrow o\ ;\ \}$
  assert $\mathit{Legal}[\mathit{Inv}_T(o)] \wedge \mathit{Inv}_T(o)$;
  $o.\mathit{inv} \leftarrow \mathit{true}$

$\mathbf{unpack}_T\ o \ \equiv$
  assert $o \neq \mathit{null} \wedge o.\mathit{owner} = \mathbf{tid} \wedge o.\mathit{inv}$;
  $o.\mathit{inv} \leftarrow \mathit{false}$;
  foreach $(f \in \mathit{RepFields}_T$ **where** $o.f \neq \mathit{null})$
    $\{\ o.f.\mathit{owner} \leftarrow \mathbf{tid}\ ;\ \}$

acquire $o \ \equiv$
  assert $o \neq \mathit{null} \wedge o.\mathit{owner} \neq \mathbf{tid}$;
  await $(o.\mathit{owner} = \mathit{null})\ \{\ o.\mathit{owner} \leftarrow \mathbf{tid}\ ;\ \}$

release $o \ \equiv$
  assert $o \neq \mathit{null} \wedge o.\mathit{owner} = \mathbf{tid} \wedge o.\mathit{inv}$;
  $o.\mathit{owner} \leftarrow \mathit{null}$

$o.f := v \ \equiv$
  assert $o \neq \mathit{null} \wedge o.\mathit{owner} = \mathbf{tid} \wedge \neg o.\mathit{inv}$;
  $o.f \leftarrow v$

$x := E \ \equiv$
  assert $\mathit{Legal}[E]$;
  $x \leftarrow E$

In the above, we write $\mathit{Legal}[E]$ to denote the predicate that says that every access expression in E is transitively owned by the current thread, as stipulated by the last bullet above. In particular,

$$\mathit{Legal}[x] \ \equiv\ \mathit{true}$$
$$\mathit{Legal}[E_0\ \mathbf{op}\ E_1] \ \equiv\ \mathit{Legal}[E_0] \wedge \mathit{Legal}[E_1]$$
$$\mathit{Legal}[o.f_1.\cdots.f_n.g] \ \equiv\ o.\mathit{owner} = \mathbf{tid} \wedge o.f_1.\mathit{owner} = o \wedge \cdots \wedge o.f_1.\cdots.f_n.\mathit{owner} = o.f_1.\cdots.f_{n-1}$$

When a thread attempts to execute a statement **await** $(P)\ \{\ S\ \}$, it blocks until the condition P is true, at which point the statement S is executed; the evaluation of P that finds P to be true and the execution of S are performed as one indivisible action.

[2] In this text, threads are not objects. In some languages, like Java and C#, a thread has a representation as an object; we can avoid ambiguity in these languages by requiring that thread objects have no **rep** fields, which allows us to stipulate that when a thread object appears as an owner, it denotes the thread, not the object.

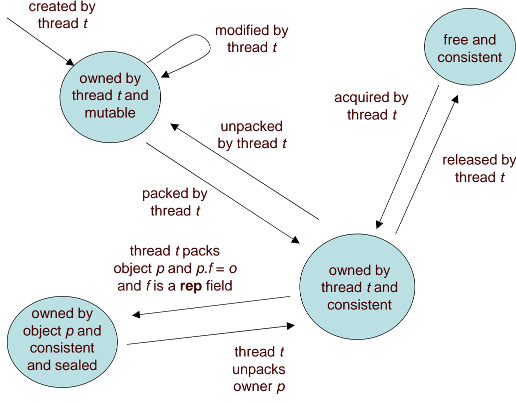

Figure 2: Object lifecycle for an arbitrary object o.

```
class Account {
  void Deposit(int amount)
    requires owner = tid ∧ inv ;
    ...
}

class IntList {
  void Add(int value)
    requires owner = tid ∧ inv ;
    ...
}
```

Figure 3: The example classes $Account$ and $IntList$, revised for their use in a multithreaded environment.

Now let us extend our running example, so that we can verify it in a multithreaded environment. First, we have to make sure that $Account$ and $IntList$ objects are accessed only when they are owned by the current thread. We choose to relegate the responsibility of exclusion to the client, a pattern which is often called *client-side locking*. We indicate this by including the requirement $owner = \textbf{tid}$ in the preconditions for methods $Deposit$ and Add, see Figure 3. A program is allowed to mention $o.owner$ only in the form $o.owner = \textbf{tid}$ and only in method contracts.

We extend the example with a $Bank$ class, which allows transfers between different accounts, see Figure 4. The method $Transfer$ requires that the $from$ and to accounts are owned by the current thread. If the precondition holds, $Transfer$ performs the intended account operations without blocking. The method $Transaction$ does the same thing, but has no requirement on thread ownership. Therefore, $Transaction$ acquires the $from$ and to objects, each of which might block.

Note that method $Transfer$ declares a postcondition, whereas method $Transaction$ does not. In fact, $Transaction$ cannot ensure the same postcondition as $Transfer$, since other threads might intervene as soon as the account objects are released. For method $Transfer$, on the other hand, the postcondition is stable, since the calling thread owns the account objects, which affords it exclusive access.

Our methodology ensures the following program invariant, for

```
class Bank {
  static void Transfer(Account from,
                       Account to,
                       int amount)
    requires from ≠ null ∧ to ≠ null ∧ from ≠ to ∧
             from.owner = tid ∧ to.owner = tid ;
    ensures from.bal = old(from.bal) − amount ∧
            to.bal = old(to.bal) + amount ;
  {
    from.Deposit(−amount) ;
    to.Deposit(amount) ;
  }
  static void Transaction(Account from,
                          Account to,
                          int amount)
    requires from ≠ null ∧ to ≠ null ∧ from ≠ to ;
  {
    acquire from ;
    acquire to ;
    Transfer(from, to, amount) ;
    release to ;
    release from ;
  }
}
```

Figure 4: A safe multithreaded bank example.

each class T:

PROGRAM INVARIANT 2.

$$(\forall\, o\colon T \bullet\ o.inv \implies Inv_T(o)) \tag{0}$$

$$(\forall\, f \in RepFields_T, o\colon T \bullet\ o.inv \implies o.f = null \lor o.f.owner = o) \tag{1}$$

$$(\forall\, o\colon T \bullet\ o.owner \notin \mathfrak{thread} \implies o.inv) \tag{2}$$

4. SOUNDNESS

In this section, we prove two results for our methodology. First, there are no data races. Second, if an object is consistent, its invariant holds.

A *data race* occurs when a field is accessed concurrently by two threads and at least one of the threads is performing a write to the field. If a data race occurs, the values read or written by a thread may be unpredictable, which severely complicates reasoning about the program.

As we have formalized our methodology in the previous section, there actually are data races, in particular on the $owner$ field. Fortunately, we can eliminate these data races by introducing redundant thread-local data into our program state, as follows:

- With each thread t, we associate a thread-local table $owns$, which maps object references to booleans.
- We extend the semantics of all statements that perform updates on $owner$ fields so that they also update the local thread's $owns$ variable. These updates will maintain the following invariant, for any object o and thread t:

$$t.owns[o] \implies o.owner = t$$

- We modify the semantics of all statements whose preconditions require $o.owner = \textbf{tid}$ for some o, so that these preconditions instead require $\textbf{tid}.owns[o]$.

- We assume any write to the *owner* field of an object to be an indivisible action.

With these modifications, we can now prove the following lemma and theorem:

LEMMA 0. *The methodology guarantees that (1) holds in all reachable states.*

THEOREM 1 (RACE FREEDOM). *Consider any object o in an execution of a program. If t is a thread that transitively owns o, then t is the only thread that can read or write a field of o or change the transitive ownership of o. Furthermore, if the transitive owner of o is null, then the only field of o that a thread reads or writes is $o.owner$, and the thread reads and writes $o.owner$ only at a moment when $o.owner = null$.*

We prove Lemma 0 and Theorem 1 together:

PROOF. Consider an arbitrary execution of the program. We prove by induction that the required properties hold in every prefix of the execution.

We look at our formalization of each program construct, as given in the previous section. Except for the **unpack** and **acquire** statement, these rules guarantee that each read or write of a field $o.f_1.\cdots.f_n.g$ is protected by an expression equivalent to the expansion of $Legal[o.f_1.\cdots.f_n.g]$ (we assume the evaluation of $\wedge$ to be conditional-and). By the induction hypothesis, these conditions are stable (with respect to the execution of other threads).

This property is also guaranteed for the **unpack** statement, except for its update of $o.f.owner$. Here's where we need the lemma. By the inductive hypothesis of the lemma, we have the disjunction $o.f = null \vee o.f.owner = o$ immediately after checking $o.inv$. By the inductive hypothesis of the theorem, this disjunction is stable. Therefore, $o.f.owner = o$ holds inside the **foreach** loop (unless a previous iteration of the **foreach** loop has already assigned **tid** to $o.f.owner$, which is also okay; this situation arises if o has two **rep** fields referencing the same part).

For the **acquire** statement, the reading and writing of $o.owner$ happens at a time when $o.owner = null$, as required by the theorem.

For the lemma, (1) holds in the empty prefix of the execution, since no objects are allocated then, which means the quantifications are vacuously true. We now turn to nonempty prefixes of the execution.

Condition (1) can be violated if the quantifier's range is enlarged to a newly allocated object. But new objects are initially mutable, so (1) is maintained.

Condition (1) can be violated if an inv field is set to $true$, which happens only in the **pack** statement. There, the update of $o.inv$ is preceded by assignments to $o.f.owner$ for representation fields $o.f$. By the theorem, the effect of these assignments is stable, and thus **pack** maintains (1).

Condition (1) can also be violated if a representation field $o.f$ is changed to a non-null value when $o.inv$ holds. But only the field update statement writes to fields, and its update is protected by $\neg o.inv$, which by the theorem is stable.

Finally, condition (1) can be violated if $p.owner$ is changed to a value q, when there is an object r and representation field g such that

$$r \neq q \wedge r.inv \wedge r.g = p$$

for then, after the assignment, we would have

$$r.inv \wedge r.g \neq null \wedge r.g.owner = q$$

The assignment to $o.f.owner$ in the **pack** statement is okay, because we argue that there are no r and g such that $r.g = o.f \wedge r.inv$: For a contradiction, suppose there are such an r and g. Then, by the induction hypothesis of (1), $r.g = null \vee r.g.owner = r$. It can't be $r.g = null$, because $o.f \neq null$. And it can't be $r.g.owner = r$, because the **pack** statement checks $o.f.owner$ to be a thread, not the object r.

The **unpack** statement changes $o.f.owner$, so we again argue that there are no r and g such that $r.g = o.f \wedge r.inv$. At the time the **unpack** statement checks $o.inv$, the induction hypothesis of (1) tells us that $o.f = null \vee o.f.owner = o$ for all representation fields f. The update of $o.f.owner$ happens only if $o.f \neq null$, so if $o.f.owner$ is updated, then $o.f.owner$ starts off as o. So the only r in danger is o itself. But at the time of the update of $o.f.owner$, $o.inv$ is $false$.

The **acquire** statement changes $o.owner$, but does so from a state where $o.owner$ is $null$.

The **release** statement changes $o.owner$, but does so from a state where $o.owner$ is a thread, not an object. □

Because of Theorem 1, we no longer have to argue about race conditions. That is, in the proof of the Soundness Theorem below, we can assume values to be stable.

THEOREM 2 (SOUNDNESS). *The methodology guarantees that Program Invariant 2 holds in all reachable states.*

PROOF. Lemma 0 already proves (1), so it remains to prove (0) and (2).

Consider an arbitrary execution of the program. We prove by induction that Program Invariant 2 holds in every prefix of the execution.

Program Invariant 2 holds in the empty prefix of the execution, since no objects are allocated then, which means the quantifications are vacuously true.

Consider any prefix of the execution leading to a state in which Program Invariant 2 holds. Let t be the thread that is about to execute the next atomic action. We prove by case analysis that this action maintains Program Invariant 2. In all cases, we make use of the fact that the *owner* field is not mentioned in invariants.

- **Case** creation of a new object o. This operation affects only quantifications over objects, since the operation enlarges the range of such quantifications. Since $o.owner = t$ and $\neg o.inv$, and since for all p, $Inv_T(p)$ does not mention quantifications over objects, all conditions are trivially satisfied.

- **Case pack**$_T$ o. (0) and (2) follow directly from the semantics.

- **Case unpack**$_T$ o. (0) and (2) follow directly from the semantics.

- **Case acquire** o. (0) is vacuously maintained. (2) follows directly from the semantics.

- **Case release** o. (0) is vacuously maintained. (2) follows directly from the semantics.

- **Case** $o.f := v$. (2) is vacuously maintained. We prove the maintenance of (0) for an arbitrary object p of a type T. Suppose for a contradiction that $p.inv$ holds and that $Inv_T(p)$ depends on $o.f$. Then o must be reachable from p via non-null **rep** fields. Through repeated application of (1) and (2), we obtain that $o.inv$ holds. This contradicts the action's precondition, which incorporates $\neg o.inv$.

This concludes the proof. □

Having proved the Soundness Theorem, we can simplify the definition of $Legal$. In particular, we only need to check that the current thread owns the root object of an access expression and that all fields in the intermediate dereferences in the access expression are **rep** fields:

$$\begin{array}{l} Legal[o.f_1.\cdots.f_n.g] \;\equiv \\ \quad o.owner = \mathbf{tid} \\ \quad \text{and } f_1,\ldots,f_n \text{ are all } \mathbf{rep} \text{ fields} \end{array}$$

Program invariant (1) takes care of the rest.

The soundness proof assumes an interleaving semantics. This implies that memory accesses are sequentially consistent. Sequential consistency means that there is a total order on all memory accesses, such that each read action yields the value written by the last write action.

Unfortunately, most execution platforms do not actually guarantee sequential consistency. However, many do guarantee the following property, see for instance Manson and Pugh's proposed memory model for Java [14]:

> If in all sequentially consistent executions of a program P, all conflicting accesses are ordered by the happens-before relation, then all executions of P are sequentially consistent.

Since Theorem 1 proves the absence of data races, our Soundness Theorem is relevant even in these systems, provided a happens-before edge exists between writing the $owner$ field in the **release** statement and reading the $owner$ field in the **acquire** statement.

5. STATIC VERIFICATION

Our Soundness Theorem proves three properties that hold in every reachable state. These properties can therefore be assumed by a static program verifier at any point in the program.

By Theorem 1, we know that the values read by a thread are stable with respect to other threads. That is, as long as an object remains in the thread's ownership domain, the fields of the object are controlled exactly in the same way that fields of objects are controlled in a sequential program. Therefore, static verification proceeds as for a sequential program.

For objects outside the thread's ownership domain, all bets are off (as we alluded to in the discussion of the $Transaction$ method in Figure 4). But since a thread cannot read fields of such objects (Theorem 1), static verification is unaffected by the values of those fields.

When an object o enters a thread's ownership domain, we know that the invariants of all objects in o's ownership domain hold. In particular, due to our non-reentrant **acquire** statement and program invariant (2) of the Soundness Theorem, we have $o.inv$. To model the intervention of other threads between exclusive regions, a static verifier plays havoc on the fields of all objects in o's ownership domain after each **acquire** o operation. The static verifier can then assume $o.inv$. By repeated applications of program invariants (1) and (2), the verifier infers $p.inv$ for all other objects p in the ownership domain of o. Thus, by program invariant (0), the verifier infers that the invariants of all of these objects hold.

To check our methodology at run time, we only need to check the assertions prescribed in Section 3. However, to reason modularly about a program, as in static modular verification, one needs method contracts. We have already seen examples of pre- and postconditions, but method contracts also need to include *modifies clauses*, which frame the possible effects a method can have within the thread's ownership domain, see [1].

```
public class AcqRel {
  private boolean free ;
  public final synchronized void acquire()
    { while (!free) { wait() ; } free = false ; }
  public final synchronized void release()
    { free = true ; notify() ; }
}
```

Figure 5: Example implementation of acquire and release in Java.

6. SAFE CONCURRENCY IN JAVA AND C#

Our methodology uses **acquire** and **release** as synchronization primitives. But how, if at all, does this apply to the *synchronized* of Java (or, equivalently, C#'s **lock** statement)? One might think that it would suffice to map Java's **synchronized** statement to **acquire** and **release** statements as follows:

$$\begin{array}{l} [\![\, \mathbf{synchronized}\ (o)\ \{\ S\ \}\ ]\!] = \\ \quad \mathbf{acquire}\ o\ ; \\ \quad \mathbf{try}\ \{\ S\ \}\ \mathbf{finally}\ \{\ \mathbf{release}\ o\ ;\ \} \end{array}$$

Unfortunately, this approach is incorrect. Specifically, entering a **synchronized** statement is not semantically equivalent to the **acquire** statement because Java considers an object to be initially not owned, whereas our methodology considers an object to be initially owned by the thread that creates it. This manifests itself in the following specific behavior: in Java, the first thread that attempts to enter a synchronized statement always succeeds immediately; in our methodology, a **release** operation must occur on an object before any thread can successfully acquire it, even the first time.

Additionally, in this approach there is no syntax for an object's initial release operation; as a result, an object could never become free. One might suggest having an implicit release operation when an object is created, and requiring even the creating thread to synchronize on the object, even in the object's constructor. But this is problematic, since it would not give the creating thread a chance to establish the object's invariant before it is released.

But there are at least two ways to achieve a correct mapping between our methodology and Java and C#. The first consists of implementing $acquire$ and $release$ methods on top of the language's built-in primitives. An example implementation in Java is shown in Figure 5. With this implementation, acquiring an object o would correspond to calling the $acquire$ method of the $AcqRel$ object associated with object o. The latter association could be achieved using *e.g.* a hash table, or, depending on platform constraints, more efficient methods, such as merging the $AcqRel$ class into class $Object$.

The second way to apply our methodology to Java and C#, is by modifying the methodology. Specifically, a modified methodology exists such that executing an **acquire** or **release** statement on an object corresponds exactly with entering or exiting a **synchronized** statement that synchronizes on the object. The modification involves the introduction of an additional boolean field, called *shared*, in each object. The field is initially *false*, it can be mentioned only in method contracts, and it can be updated only through a special **share** statement.

In the modified methodology, the semantics of the statements

share, **acquire**, and **release** are as follows:

$\textbf{acquire}\ o\ \equiv$
$\quad\textbf{assert}\ o \neq \mathit{null} \wedge o.\mathit{shared} \wedge o.\mathit{owner} \neq \textbf{tid}\ ;$
$\quad\textbf{await}\ (o.\mathit{owner} = \mathit{null})\ \{\ o.\mathit{owner} \leftarrow \textbf{tid}\ ;\ \}$

$\textbf{release}\ o\ \equiv$
$\quad\textbf{assert}\ o \neq \mathit{null} \wedge o.\mathit{owner} = \textbf{tid} \wedge o.\mathit{shared} \wedge o.\mathit{inv}\ ;$
$\quad o.\mathit{owner} \leftarrow \mathit{null}$

$\textbf{share}\ o\ \equiv$
$\quad\textbf{assert}\ o \neq \mathit{null} \wedge o.\mathit{owner} = \textbf{tid} \wedge \neg o.\mathit{shared} \wedge o.\mathit{inv}\ ;$
$\quad o.\mathit{owner} \leftarrow \mathit{null}\ ;$
$\quad o.\mathit{shared} \leftarrow \mathit{true}$

In the modified methodology, exclusive access to an object by its creating thread during initialization is ensured not through run-time synchronization, but through constraints on the the newly introduced *shared* field imposed by the methodology.

7. RUN-TIME CHECKING

Our methodology supports both static verification and run-time checking. The advantage of static verification is that it decides the correctness of the program for all possible executions, whereas run-time checking decides whether the running execution complies with the methodology. The disadvantage of static verification is that it requires method contracts, including preconditions, postconditions, and modifies clauses, whereas run-time checking does not.

If a program has been found to be correct through static verification, no run-time checks would ever fail and they can be omitted. When running a program without run-time checks, the only run-time cost imposed by our methodology is the implementation of the **acquire** and **release** statements (as in Figure 5, for example); none of the fields or other data structures introduced by our methodology need to be present, and none of the **assert** statements need to be executed. In particular, the **pack** and **unpack** statements become no-ops.

For run-time checking, two fields, the *inv* field and the *owner* field, need to be inserted into each object. To prove race freedom, we eliminated the races on the *owner* fields by introducing an *owns* table for each thread; however, on most platforms, including Java and C#, these races are in fact benign and the *owns* tables can be omitted.

8. RELATED WORK

The Extended Static Checkers for Modula-3 [6] and for Java [8] attempt to statically find errors in object-oriented programs. These tools include support for the prevention of data races and deadlocks. For each field, a programmer can designate which lock protects it. However, these two tools trade soundness for ease of use; for example, they do not take into consideration the effects of other threads between regions of exclusion. Moreover, various engineering trade-offs in the tools notwithstanding, the methodology used by the tools was never formalized enough to allow a soundness proof.

Method specifications in our methodology pertain only to the pre-state and post-state of method calls. Some systems [18, 9] additionally support specification and verification of the atomic transactions performed during a method call, even though this information does not translate into knowledge about the post-state (because of intervening transactions by other threads).

A number of type systems have been proposed that prevent data races in object-oriented programs. For example, Boyapati *et al.* [4] parameterize classes by the protection mechanism that will protect their objects against data races. The type system supports thread-local objects, objects protected by a lock (*i.e.*, another object), read-only objects, and unique pointers. However, the ownership relationship that relates objects to their protection mechanism is fixed. Also, their type system does not support object invariants.

Quite similar to ours is the methodology used by Vault (*cf.* [5]), which can be applied in a concurrent setting. In Vault, linear types guarantee that objects are owned by a single thread only. The **pack** and **unpack** operations are implicit in Vault. The **acquire** operation is not supported, because the object to be acquired may have been deleted; however, it would be possible to add the **release acquire** operation pair to a version of Vault for a garbage-collected language. Vault's methodology is enforced by a static type system, which has advantages but limits its supported invariants. For example, Vault supports neither general predicates on the fields of an object nor relations on the fields of more than one object in an aggregate.

Atomizer [7] dynamically checks the atomicity of unannotated methods. It ensures that all statements in the method can be reasoned about sequentially. However, Atomizer does not easily support atomicity at different abstraction levels, which our methodology does.

Ábrahám-Mumm *et al.* [0] propose an assertional proof system for Java's reentrant monitors. It supports object invariants, but these can depend only on the fields of **this**. No claim of modular verification is made.

The rules in our methodology that an object must be consistent when it is released, and that it can be assumed to be consistent when it is acquired, are taken from Hoare's work on monitors and monitor invariants [10].

There are also tools that try dynamically to detect violations of safe concurrency. A noteable example is Eraser [19]. It finds data races by looking for locking-discipline violations. The tool has been effective in practice, but does not come with guarantees about the completeness nor the soundness of the method.

The basic object-invariant methodology that we have built on [1] has also been extended in other ways for sequential programs [13, 3, 12].

9. CONCLUSIONS

Our new sound, modular, and simple locking methodology helps in defining leak-proof ownership domains. Several aspects of this new approach are noteworthy. First, sequentially verifiable programs are race free. Due to the necessary preconditions for reading and writing, only one thread at a time can access the objects of an ownership domain. Second, the owner of an object can change over time. In particular, an object may move between ownership domains. Third, our methodology can be efficient; it acquires only one lock per ownership domain, where the domain consists of many objects. Further, at run time, we only need to keep track of a bit per object that says whether or not there exists a thread that transitively owns the object.

We are in the process of adding support for this methodology to Spec#, an extension of C# with contracts [2]. Spec# performs both run-time checking and static verification (via the program verifier Boogie).

But there is obviously much more left to be done. One important area of work is the assessment and optimization of the efficiency of both static verification and run-time checking on realistic examples. Also, we are currently extending the approach to deal with other design patterns, like traversals, wait and notification, condition variables, multiple reader writers, *etc.* In fact, our ambition is

to cover many of the design patterns described by Doug Lea [11]. Another area of future work is the treatment of liveness properties, such as deadlock freedom.

Since our methodology is an extension of an object-invariant methodology for sequential programs, it would be interesting to automatically infer for given sequential programs the additional contracts necessary for concurrency.

Acknowledgments. We thank Manuel Fähndrich, Tony Hoare, the members of the Boogie team, and the referees for insightful remarks and suggestions.

10. REFERENCES

[0] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Verification for Java's reentrant multithreading concept. In *Foundations of Software Science and Computation Structures, 5th International Conference, FoSSaCS 2002*, volume 2303 of *Lecture Notes in Computer Science*, pages 5–20. Springer, April 2002.

[1] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[2] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Start devices (CASSIS)*, Lecture Notes in Computer Science. Springer, 2004. To appear.

[3] Mike Barnett and David Naumann. Friends need a bit more: Maintaining invariants over shared state. In Dexter Kozen, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science, pages 54–84. Springer, July 2004.

[4] 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.

[5] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 36, number 5 in *SIGPLAN Notices*, pages 59–69. ACM, May 2001.

[6] 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.

[7] Cormac Flanagan and Stephen N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the 2004ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004)*, volume 39, number 1 in *SIGPLAN Notices*, pages 256–267. ACM, January 2004.

[8] 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.

[9] Stephen N. Freund and Shaz Qadeer. Checking concise specifications for multithreaded software. *Journal of Object Technology*, 3(6):81–101, June 2004.

[10] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[11] Doug Lea. *Concurrent Programming in Java*. Addison Wesley, 2000.

[12] K. Rustan M. Leino and Peter Müller. Modular verification of global module invariants in object-oriented programs. Technical Report 459, ETH Zürich, 2004.

[13] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Martin Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.

[14] Jeremy Manson and William Pugh. Requirements for a programming language memory model. Workshop on Concurrency and Synchronization in Java Programs, in association with PODC, July 2004.

[15] 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.

[16] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *ECOOP'98—Object-oriented Programming: 12th European Conference*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer, July 1998.

[17] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[18] Shaz Qadeer, Sriram K. Rajamani, and Jakob Rehof. Summarizing procedures in concurrent programs. In *Proceedings of the 2004 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004)*, volume 39, number 1 in *SIGPLAN Notices*, pages 245–255. ACM, January 2004.

[19] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997. Also appears in *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pages 27–37, Operating System Review 31(5), 1997.