Safe Concurrency for Aggregate Objects with Invariants

Bart Jacobs[0] K. Rustan M. Leino[1] Frank Piessens[0] Wolfram Schulte[1]

[0]Dept. of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium
{bartj,frank}@cs.kuleuven.be

[1]Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
{leino,schulte}@microsoft.com

Abstract

Developing safe multithreaded software systems is difficult due to the potential unwanted interference among concurrent threads. This paper presents a flexible methodology for object-oriented programs that protects object structures against inconsistency due to race conditions. It is based on a recent methodology for single-threaded programs where developers define aggregate object structures using an ownership system and declare invariants over them. The methodology is supported by a set of language elements and by both a sound modular static verification method and run-time checking support. The paper reports on preliminary experience with a prototype implementation.

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.

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

```
void Transfer(DualAccounts o, int amount) {
    int a = o.a ; / 0 / o.a := a - amount ;
    int b = o.b ; / 1 / o.b := b + amount ;
}
```

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, consider two threads that both perform a transfer on the same dual accounts. If one transfer occurs either at point 0 or 1 during the other transfer, then the invariant is not 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 greatly. 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 [24].

In this paper, we present a new programming methodology for 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 sound modular static verification by requiring methods to be annotated with simple ownership requirements; as an alternative, it also provides run-time checking support that does not require any method annotations. The methodology is an extension of the Spec# methodology for sequential code, as described in our previous work [3].

We see the proposed methodology as a basis for a comprehensive approach to the specification and verification of multithreaded programs. In this paper, we focus on the core safe concurrency methodology; we are also investigating extensions for increased parallelism and deadlock prevention, but we do not consider those in this paper

The paper proceeds as follows. The next four sections

```
class DualAccounts {
  int a, b ;
  invariant a + b = 0 ;
  void Transfer(int amount)
    requires inv ;
  {
    unpack (this) ;
    int a = this.a ; this.a := a − amount ;
    int b = this.b ; this.b := b + amount ;
    pack (this) ;
  }
}
```

Figure 0. A *DualAccounts* class in the object invariant methodology of Section 1.

gradually introduce our methodology: Section 1 introduces object invariants, Section 2 introduces confinement within objects. Section 3 adds confinement within threads to offer a methodology for safe concurrency of aggregate objects with invariants. In Section 4, we show how to apply our methodology to Java and C#. In Section 5, we explain how the methodology can be enforced through static verification and through run-time checking. Sections 6 and 7 mention related work and conclude. We included a proof of soundness in an accompanying Technical Report [16].

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 [3] 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 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.[1]

$$\begin{array}{l}
\mathbf{pack}_T\ o \quad \equiv \\
\quad \mathbf{assert}\ o = null \quad \neg o.inv \quad Inv_T(o)\ ; \\
\quad o.inv \leftarrow true \\
\\
\mathbf{unpack}_T\ o \quad \equiv \\
\quad \mathbf{assert}\ o = null \quad o.inv\ ; \\
\quad o.inv \leftarrow false \\
\\
o.f := E \quad \equiv \\
\quad \mathbf{assert}\ o = null \quad \neg o.inv\ ; \\
\quad o.f \leftarrow E
\end{array}$$

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.

$$(\quad o\colon T \bullet\ o.inv = \quad Inv_T(o)\ )$$

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

2 Confinement within Objects

Consider 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 deposits recorded in the history, as is captured by the invariant. In this section, we introduce measures that allow $Account$'s invariant to mention fields of $hist$.

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

[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.

[1] The restrictions on $o.inv$ in the preconditions of **pack** and **unpack** are not necessary for soundness, but omitting them would not enable more use cases and the stricter rules are helpful to readers of annotated programs.

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

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

Figure 1. An example illustrating the aggregate objects methodology of Section 2.

fields holding representation objects with a **rep** modifier (*cf.* [23]).[2]

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 [22, 3].

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. 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.[3] We use the function $RepFields_T$ to denote the fields marked **rep** in class T.

```
unpack_T o  ≡
  assert o = null    o.owner = null    o.inv ;
  o.inv ← false ;
  foreach (f    RepFields_T  where o.f = null)
    { o.f.owner ← null ; }

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
```

For illustration purposes, let us inspect a trace of the invocation $acct.Deposit(100)$ for a non-null $Account$ object $acct$ that satisfies the precondition of $Deposit$, where we focus only on the involved inv and $owner$ fields of the involved objects. First, $Deposit$ 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 $Deposit$ 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.

$$
\begin{array}{l}
(\quad o\colon T \bullet\ o.inv = \quad Inv_T(o)\ ) \\
(\quad f \quad RepFields_T, o\colon T \bullet \\
\qquad o.inv = \quad o.f = null \quad o.f.owner = o\ ) \\
(\quad o\colon T \bullet\ o.owner = null = \quad o.inv\ )
\end{array}
$$

3 Confinement within Threads

In this section, we combine the object ownership scheme of the previous section with a simple notion of *ownership of an object by a thread*, to achieve confinement within threads.

We assume that the execution platform provides a primitive synchronization construct, in the form of $P()$ and $V()$ operations on objects, where $P()$ and $V()$ are the classic semaphore operations. We assume that the semaphores are initialized at object creation time to a closed state (*i.e.* a $P()$ operation will block). In Java and C#, the platform primitive synchronization constructs are based on monitors, but it is easy to implement $P()$ and $V()$ on top of these.

In the object ownership scheme above, objects are either part of an aggregate object or they are free, and if they

[2] A class may also have non-**rep** fields. For example, a $LinkedList$ class would store its elements in non-**rep** fields, since these elements would typically be owned by the owner of the $LinkedList$, not by the $LinkedList$ itself.

[3] This is a slightly different use of the $owner$ field than in [20].

are free it 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.

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 accesses, *i.e.*, reads or writes, one of its fields. Since no two threads can transitively own the same object concurrently, this guarantees exclusion.

The rules for ownership of objects by threads 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 $o.g$ only if it owns o, and an expression $o.f_1.\cdots.f_n.g$ only if it owns o, o is consistent, and each f_i is a **rep** field.

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 given in Figure 3, where we denote the currently executing thread by **tid**. The operations use the $P()$ and $V()$ semaphore operations to synchronize access to the $owner$ field; also, the invariant is maintained that whenever the semaphore is in the open state, the $owner$ field is $null$. The definitions use $Legal[\![E]\!]$, which is defined as follows:

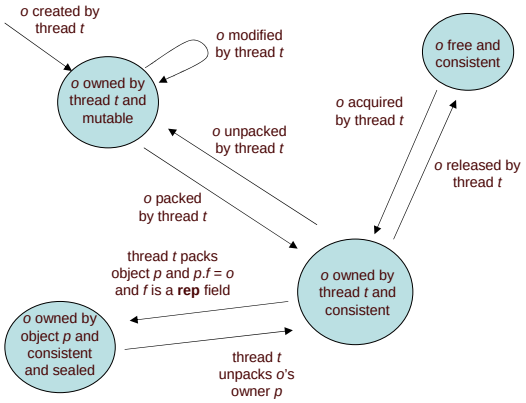

Figure 2. Object lifecycle for an arbitrary object o.

$$
\begin{array}{l}
Legal[\![x]\!] \quad \equiv \quad true \\
Legal[\![\,\mathbf{op}_k(E_0,\ldots,E_{k-1})]\!] \quad \equiv \quad \bigwedge_{0\le i<k} Legal[\![E_i]\!] \\
Legal[\![E.g]\!] \quad \equiv \quad Confined[\![E]\!] \\
Confined[\![E]\!] \quad \equiv \quad Legal[\![E]\!] \quad E = null \\
\quad (E.owner = \mathbf{tid} \\
\quad ((E \text{ has the form } E'.f) \quad E'.inv \quad f \text{ is } \mathbf{rep}))
\end{array}
$$

Now let us extend our running example, so that we can verify it in a multithreaded environment. We have to make sure that $Account$ objects are accessed only when they are owned by the current thread. Figure 4[4] illustrates two ways to achieve this, known as *client-side locking* and *provider-side locking*. Client-side locking, exemplified by method $Deposit$, means that responsibility of exclusion is relegated to the client. We indicate this by including the requirement $owner = \mathbf{tid}$ in the precondition. A program is allowed to mention the $owner$ field only in the form $o.owner = \mathbf{tid}$ and only in method contracts.

The $Transaction$ method is a case of provider-side locking. The method has no precondition, which means the client has no obligations. There is a drawback to provider-side locking, however. Whereas method $Deposit$ can guarantee that the caller will find the account's balance increased by $amount$ after the method returns, method $Transaction$ can provide no such guarantee, since by the time the caller inspects the balance (and before it can do this, it must ac-

[4]The postcondition does not repeat $owner = \mathbf{tid} \wedge inv$ because this is implied by the default modifies clause, which effectively says that only the user-declared fields of **this** (but not inv or $owner$) and the fields of transitively owned or newly allocated objects may change (see [3]).

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

unpack_T o  ≡
  assert o = null  o.owner = tid  o.inv ;
  o.inv ← false ;
  foreach (f  RepFields_T  where o.f = null)
    { o.f.owner ← tid ; }

acquire o  ≡
  assert o = null ;
  o.P();
  o.owner ← tid ;

release o  ≡
  assert o = null  o.owner = tid  o.inv ;
  o.owner ← null;
  o.V();

o.f := x  ≡
  assert o = null  o.owner = tid  ¬o.inv ;
  o.f ← v

x := E  ≡
  assert Legal[[E]] ;
  x ← E
```

Figure 3. Confinement within threads

quire the account) other threads may have performed further transactions. Therefore, client-side locking is generally preferable. In any case, our methodology ensures that either the client or the provider performs the necessary locking. Also, our methodology requires that preconditions and postconditions perform only legal field accesses, which would for example prevent method *Transaction* from declaring the postcondition $bal = \mathbf{old}(bal) + amount$.

Our methodology ensures the following program invariant, for each class T:

Program Invariant 2.

$$
\begin{array}{l}
(\ \ o\colon T \bullet\ o.inv = \ \ Inv_T(o)\ ) \\
(\ \ f \ \ RepFields_T, o\colon T \bullet \\
\quad\ o.inv = \ \ o.f = null \ \ o.f.owner = o\ ) \\
(\ \ o\colon T \bullet\ o.owner \ \ \mathfrak{thread} = \ \ o.inv\ )
\end{array}
$$

```
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 owner = tid  inv ;
    ensures bal = old(bal) + amount ;
  {
    unpack (this) ;
    hist.Add(amount) ;
    bal := bal + amount ;
    pack (this) ;
  }

  void Transaction(int amount) {
    acquire (this) ;
    Deposit(amount) ;
    release (this) ;
  }
}
```

Figure 4. *Account* **example, modified for the multithreaded methodology of Section 3.**

4 Application to Java and C#

In this section, we look at the application of our methodology to the popular concurrent object-oriented programming languages Java and C#.

Clearly, the **acquire** and **release** operations are semantically different from entering and exiting a synchronized block. Hence, there are essentially two ways to apply our approach to Java and C#: by using semaphores instead of synchronized blocks (Section 4.0) or by modifying the methodology (Section 4.1). Finally, in Section 4.2, we discuss important memory consistency issues.

4.0 Using semaphores

Perhaps the most powerful option is to use some semaphore implementation. Acquiring an object o would correspond to performing a P operation on the semaphore object associated with object o. The latter association can be achieved using *e.g.* a hash table, or, depending on platform constraints, more efficient methods, such as merging the semaphore implementation into class *Object*.

4.1 Unshared Objects

In this subsection, we present a modified methodology that maps directly to the Java **synchronized** statement and the equivalent C# **lock** statement.

In the methodology of Section 3, as soon as an object reference is leaked from its creating thread t to another thread u, u may attempt to acquire the object. It will block until t has finished its initialization and releases the object. In Java and C#, however, objects are initially free, so an attempt by u to acquire the object would succeed, causing it to interfere with the object's initialization.

In the modified methodology presented in this subsection, we protect the initial ownership of an object by its creating thread, by disallowing that other threads even attempt to acquire an object until its creating thread releases ownership of it. Specifically, we distinguish objects that have never been released from objects that have been released at least once. We call the former the *unshared* state, and the latter the *shared* state.

An object indicates its state through a new boolean $shared$ field. It is initialized to $false$ when the object is created, and it becomes $true$ upon the initial release of the object. Before attempting to acquire an object, a thread must first check the $shared$ field, and only if the field is $true$ is the thread allowed to proceed. A thread is allowed to update the $shared$ field only through the new **share** command, which sets it to $true$. It follows that a $shared$ flag, once set, is never cleared again. A thread uses this command for the initial release of an object. There are no restrictions on reading the field in program code. However, in object invariants and method contracts, a field $o.shared$ may be mentioned in an assertion $Q(o.shared)$ only if $Q(false)$ implies $Q(true)$, so that the predicate is invariant under updates of the $shared$ field.

In the absence of special measures, this system would suffer from a data race between a thread that sets an object's $shared$ field and a thread that checks it. The practical effect of this would be that even after a thread successfully acquired an object, it would not necessarily see the initializations performed on the object. To avoid this problem, we declare the $shared$ field as **volatile**.

Another feature of the Java **synchronized** statement that necessitates the methodology to be adapted is the fact that it is *re-entrant*. This is significant since an **acquire** o statement sets $o.owner \leftarrow \mathbf{tid}$. Without further adaptations, unsoundness could be obtained by acquiring an object, packing it into some other object, and then acquiring it again. We render re-entrancy impossible by (a) not allowing shared objects to become owned by other objects, and (b) asserting that $o.owner = \mathbf{tid}$ as part of an **acquire** o operation.

The modified $\mathbf{pack}_T$, **acquire** and **release** statements and the new **share** statement are defined formally in Figure 5. The **acquire** and **release** statements now use the Java **monitorenter** and **monitorexit** bytecode instructions instead of the $P()$ and $V()$ instructions. **monitorenter** and **monitorexit** correspond to entering and exiting a **synchronized** block. In the same figure we also define the **synchronized** statement, which combines the modified **acquire** and **release** statements. Apart from the additional checks that help enforce the data-race-freedom and the object invariants, this statement is equivalent to a regular **synchronized** statement.

The statements in Figure 5, combined with the state-

$\mathbf{pack}_T\ o \ \equiv$
  assert $o = null \quad o.owner = \mathbf{tid} \quad \neg o.inv$;
  assert $Legal[\![Inv_T(o)]\!] \quad Inv_T(o)$;
  foreach $(f \quad RepFields_T$ **where** $o.f = null)$ *{*
    assert $o.f.owner = \mathbf{tid} \quad o.f.inv$;
    assert $\neg o.f.shared$;
  }
  foreach $(f \quad RepFields_T$ **where** $o.f = null)$
    { $o.f.owner \leftarrow o$; *}*
  $o.inv \leftarrow true$;

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

acquire $o \ \equiv$
  assert $o = null \quad o.shared \quad o.owner = \mathbf{tid}$;
  monitorenter o ;
  $o.owner \leftarrow \mathbf{tid}$

release $o \ \equiv$
  assert $o = null \quad o.owner = \mathbf{tid} \quad o.inv$;
  assert $o.shared$;
  $o.owner \leftarrow null$;
  monitorexit o

synchronize $(o)\ S \ \equiv$
  assert $o = null \quad o.shared \quad o.owner = \mathbf{tid}$;
  synchronize (o) *{*
    $o.owner \leftarrow \mathbf{tid}$;
    S
    assert $o.owner = \mathbf{tid} \quad o.inv$;
    $o.owner \leftarrow null$;
  }

Figure 5. The modified methodology

ments in Figure 3 that remain unchanged, can now be used to annotate a plain Java program. Two methods can then be used to verify that the annotated program complies with the methodology:

- The auxiliary inv, $owner$, and $shared$ fields, the assignments to those fields, and the **assert** statements are emitted as part of the program. When this instrumented program is run, each execution either complies with the methodology or is aborted immediately when a violation is detected.
- The program, including the auxiliary fields, assignments, and **assert** statements are translated into the language of a theorem prover for sequential programs. Additional constructs are inserted at **acquire** statements to make sure the theorem prover does not assume anything about the state of the acquired aggregate object, other than that its object invariant holds. The verifier is then executed on the program. If verification succeeds, all auxiliary fields, assignments, and **assert** statements are removed from the program. All executions of the stripped program are known to comply with the methodology and therefore to be data-race-free and to maintain their object invariants.

4.2 Memory Consistency

When multiple threads access shared memory, the question arises of how writes performed by one thread affect the values yielded by reads performed by another thread. This is known as memory consistency. The reality of an ever growing gap between the speed of processors on the one hand and memory on the other hand has given rise to code transformations by compilers and processors and multiple levels of memory caches, to reduce the number of accesses to main memory. As a result, programmers cannot assume the most intuitive memory consistency model, known as sequential consistency, where there is a single total order on the field accesses performed by all threads, such that each read operation yields the value written by the most recent preceding write operation. This is exactly the requirement for using an *interleaving semantics*, where operations performed by threads are seen as transitions from one global state to another, and where a global state includes a single global heap that determines the value yielded by a read operation.

The challenge faced by a programming language designer is to devise a memory consistency model which can be implemented efficiently on existing hardware on the one hand, and which is easy for programmers to reason about on the other hand. This challenge was taken up in the first edition of the Java Language Specification [13], but the memory model specified there has since been shown to be both too weak for programmers and too strong for implementers [25]. Recently, an attempt was made to fix Java's memory model for the third edition of the Java Language Specification (JLS3) [14]. We designed our methodology to be sound under the memory model of JLS3. Specifically, in our soundness proof, we assume a property called DRF0 [2], which is guaranteed explicitly by JLS3 [14, 21].

Property 0 (DRF0)**.** *If a program is correctly synchronized, then all executions are sequentially consistent.*

This property depends on the following definition:

Definition 0 (Correctly Synchronized)**.** *A program is correctly synchronized iff all sequentially consistent executions are data-race-free.*

This definition can be rephrased as follows:

Property 1. *A program is correctly synchronized iff there is no sequentially consistent execution that includes two adjacent accesses of the same field by different threads, where at least one access is a write.*

We conclude that, in order to obtain results that are sound for Java, we are permitted to assume an interleaving semantics, provided that we prove the absence of data races. (And we may assume an interleaving semantics for the latter proof.)

5 Enforcing the Methodology

5.0 Static Verification

The methodology guarantees that Program Invariant 2 holds in each state. This invariant can therefore be assumed by a static program verifier at any point in the program.

The absence of data races implies 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, 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 the third component of Program Invariant 2, we have $o.inv$. To model the intervention of other threads between exclusive regions, a static verifier plays

havoc on (*i.e.* forgets all knowledge about) the fields of all objects in o's ownership domain after each $\mathbf{acquire}\ o$ operation. The static verifier can then assume $o.inv$. By repeated applications of the second and third components of Program Invariant 2, the verifier infers $p.inv$ for all other objects p in the ownership domain of o. Thus, by the first component, the verifier infers that the invariants of all of these objects hold.

As will be described below, 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 [3].

The flexibility of our ownership system compared to ownership type systems such as [6] might seem to make it unrealistic to hope to be able to statically verify programs written in this system. However, to the extent that the ownership types of these type systems can be encoded as assertions on the $owner$ field in our system, which definitely seems to be the case, the verification power of our system is in fact strictly greater than that of these type systems, due to our use of a general-purpose theorem prover, as opposed to a type checker with a fixed set of inference rules. In fact, our system can be used as a very flexible system for proving just the absence of data races, by not declaring any object invariants.

5.1 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 semaphore operations; none of the fields or other data structures introduced by our methodology need to be present, and none of the $\mathbf{assert}$ statements need to be executed. In particular, the $\mathbf{pack}$ and $\mathbf{unpack}$ statements become no-ops.

For run-time checking, two fields, the inv field and the $owner$ field, need to be inserted into each object. Checking $o.owner = \mathbf{tid}$ does introduce a data race, but this is a benign one in both Java and C#. Storing both references to objects and thread identifiers (which in Java are references to objects of class Thread) in a single field is ambiguous in principle, but this ambiguity can be resolved by requiring that Thread objects have no $\mathbf{rep}$ fields.

5.2 Preliminary Experience

We have implemented our methodology as an extension to the compiler and the static verifier of the Spec# programming language research project [4, 29] developed at Microsoft Research. Spec# extends C# with method contracts, non-null types, checked exceptions, and other reliability features.

We have applied our methodology to a few small test programs and verified them using both run-time checking and static verification. The annotation overhead is reasonable.

We are working to extend the toolset with additional features, such as support for arrays and specifications for the system libraries, so as to be able to gain experience with larger, more realistic programs.

6 Related Work

The Extended Static Checkers for Modula-3 [8] and for Java [10] 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 tradeoffs 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 [26, 12] additionally support specification and verification of the atomic transactions performed during a method call. We focus on verification of object invariants, which does not require such specifications.

A number of type systems have been proposed that prevent data races in object-oriented programs. For example, Boyapati *et al.* [6] 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 (its own lock or its root owner's lock), read-only objects, and unique pointers. However, the ownership relationship that relates objects to their protection mechanism is fixed. Also, the type system does not support object invariants.

Quite similar to ours is the methodology used by Vault (*cf.* [7]), 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.

We enable sequential reasoning and ensure consistency of aggregate objects by preventing data races. Some authors propose pursuing a different property, called *atomicity*, either through dynamic checking [9] or by way of a type system [11]. An atomic method can be reasoned about sequentially. However, we enable sequential reasoning even for non-atomic methods, by assuming only the object invariant for a newly acquired object (see Section 5.0). Also, in [11] the authors claim that data-race-freedom is unnecessary for sequential reasoning. It is true that some data races are benign, even in the Java and C# memory models; however, the data races allowed in [11] are generally not benign in these memory models; indeed, the authors prove soundness only for sequentially consistent systems, whereas we prove soundness for the Java memory model, which is considerably weaker.

Another effort based on atomicity, and perhaps the most closely related work to ours is the work on extending the Java Modeling Language to support multithreaded programs [27]. As in our methodology, there is support for object invariants and aggregate objects. A distinction is that [27] advocates marking methods as *atomic* or *independent*. Both annotations imply that the method can be reasoned about as if it was executing in isolation. We are not inclined to adopt similar features, since we feel that these properties do not abstract away sufficiently the non-observable internal behavior of the method. For example, a call to a sorting routine of a private data structure is not atomic if the implementation of the sorting routine is done in parallel using multiple threads, an implementation decision of the sorting routine that we would rather not have impacting the reasoning of the caller. We allow sequential reasoning about methods and method calls without restricting internal locking behavior, by weakening the proof rules appropriately (see Section 5.0). Another distinction is the type of ownership system used. The type system used by [27] has a fixed notion of where objects are packed and unpacked (to use our terminology), whereas our theorem prover-based system allows more flexibility. A general observation is that [27] seems to strive to be able to verify as many existing programs as possible, whereas our aim is rather to develop a methodology that prescribes how new programs should be written.

Ábrahám-Mumm *et al.* [1] 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 [15].

There are also tools that try dynamically to detect violations of safe concurrency. A notable example is Eraser [28]. 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 [3] has also been extended in other ways for sequential programs [20, 5, 19].

In the straightforward implementation proposed in this paper, mutual exclusion is achieved through coarse-grained locking. However, the methodology allows one to use other semantically equivalent techniques that may be more appropriate for particular contention patterns, while preserving the same reasoning framework and safety guarantees. Possible alternatives include fine-grained locking of the objects within an ownership domain, or a form of optimistic concurrency, such as transactional monitors [30].

7 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, it allows one to obtain data-race-freedom through sequential reasoning. 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 requires only one lock acquisition per ownership domain, even when 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 the corresponding semaphore is open or closed, *i.e.* whether the object is free on the one hand, or owned by a thread or another object on the other hand.

We have implemented support for this methodology as an extension of Spec#, which is itself an extension of C# with contracts [4]. Spec# performs both run-time checking and static verification.

Much more work remains 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 have begun extending the approach to deal with other design patterns, like traversals, wait and notification, condition variables, multiple reader writers, fine-grained locking, *etc.* In fact, our ambition is to cover many of the design patterns described by Doug Lea [18]. 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, and the members of the Boogie team for insightful remarks and suggestions. We are also grateful for the feedback we received from presenting a previous version of this paper at the SAVCBS workshop [17]. Bart Jacobs is a Research Assistant of the Research Foundation - Flanders (FWO - Vlaanderen).

References

[1] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Verification for Java's reentrant multithreading concept. In *FoSSaCS 2002*, volume 2303 of *LNCS*, pages 5–20. Springer, April 2002.

[2] Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. In *ICSA 1990*, pages 2–14. IEEE Computer Society Press, June 1990.

[3] 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.

[4] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS*, volume 3362 of *LNCS*. Springer, 2004.

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

[6] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA 2002*, volume 37, number 11 in *SIGPLAN Notices*, pages 211–230. ACM, November 2002.

[7] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI 2001*, volume 36, number 5 in *SIGPLAN Notices*, pages 59–69. ACM, May 2001.

[8] 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.

[9] Cormac Flanagan and Stephen N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *POPL 2004*, volume 39, number 1 in *SIGPLAN Notices*, pages 256–267. ACM, January 2004.

[10] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI 2002*, volume 37, number 5 in *SIGPLAN Notices*, pages 234–245. ACM, May 2002.

[11] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI 2003*, pages 338–349. ACM, 2003.

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

[13] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[14] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification (3rd Edition)*. Addison-Wesley, 2005.

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

[16] Bart Jacobs, K. Rustan M. Leino, and Frank Piessens en Wolfram Schulte. Safe concurrency for aggregate objects with invariants: Soundness proof. Technical Report MSR-TR-2005-85, Microsoft Research, jun 2005.

[17] Bart Jacobs, K. Rustan M. Leino, and Wolfram Schulte. Verification of multithreaded object-oriented programs with invariants. In Mike Barnett, Stephen H. Edwards, Dimitra Giannakopoulou, Gary T. Leavens, and Natasha Sharygina, editors, *SAVCBS 2004 Workshop Proceedings*, 2004. Technical Report 04-09, Computer Science, Iowa State University.

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

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

[20] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Martin Odersky, editor, *ECOOP 2004*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.

[21] Jeremy Manson, William Pugh, and Sarita Adve. The Java Memory Model. In *POPL 2005*, volume 40, number 1 in *SIGPLAN Notices*, pages 378–391. ACM, January 2005.

[22] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002. PhD thesis, FernUniversität Hagen.

[23] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *ECOOP'98*, volume 1445 of *LNCS*, pages 158–185. Springer, July 1998.

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

[25] William Pugh. Fixing the Java memory model. In *ACM Java Grande Conference*, June 1999.

[26] Shaz Qadeer, Sriram K. Rajamani, and Jakob Rehof. Summarizing procedures in concurrent programs. In *POPL 2004*, volume 39, number 1 in *SIGPLAN Notices*, pages 245–255. ACM, January 2004.

[27] Edwin Rodríguez, Matthew Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, and Robby. Extending sequential specification techniques for modular specification and verification of multithreaded programs. In *ECOOP 2005*, July 2005. To appear.

[28] 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.

[29] Spec# project web page. URL: http://research.microsoft.com/specsharp/.

[30] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Transactional monitors for concurrent objects. In *ECOOP 2004*, volume 3086 of *LNCS*, June 2004.