A Statically Verifiable Programming Model for Concurrent Object-Oriented Programs

Bart Jacobs[1⋆], Jan Smans[1⋆], Frank Piessens[1], Wolfram Schulte[2]

[1]DistriNet, Dept. Computer Science, K.U.Leuven
Celestijnenlaan 200A, 3001 Leuven, Belgium
{bartj,jans,frank}@cs.kuleuven.be

[2]Microsoft Research
One Microsoft Way, Redmond, WA, USA
schulte@microsoft.com

Abstract. Reasoning about multithreaded object-oriented programs is difficult, due to the non-local nature of object aliasing, data races, and deadlocks. We propose a programming model that prevents data races and deadlocks, and supports local reasoning in the presence of object aliasing and concurrency. Our programming model builds on the multithreading and synchronization primitives as they are present in current mainstream languages. Java or C# programs developed according to our model can be annotated by means of stylized comments to make the use of the model explicit. We show that such annotated programs can be formally verified to comply with the programming model. In other words, if the annotated program verifies, the underlying Java or C# program is guaranteed to be free from data races and deadlocks, and it is sound to reason locally about program behavior. We have implemented a verifier for programs developed according to our model in a custom build of the Spec# programming system, and have validated our approach on a case study.

1 Introduction

Writing correct multithreaded software in mainstream languages such as Java or C# is notoriously difficult. The non-local nature of object aliasing, data races, and deadlocks makes it hard to reason about the correctness of such programs. Moreover, many assumptions made by developers about concurrency are left implicit. For instance, in Java, many objects are not intended to be used by multiple threads, and hence it is not necessary to perform synchronization before accessing their fields. Other objects are intended to be shared with other threads and accesses should be synchronized, typically using locks. However, the program text does not make explicit if an object is intended to be shared, and as a

⋆ Bart Jacobs and Jan Smans are Research Assistants of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.-Vlaanderen)

consequence it is practically impossible for the compiler or other static analysis tools to verify if locking is performed correctly.
The contributions of this paper are as follows:

- We propose a programming model for concurrent programming in Java-like languages, and the design of a set of program annotations that make the use of the programming model explicit. For instance, a developer can annotate his code to make explicit whether an object is intended to be shared with other threads or not. These annotations provide sufficient information to static analysis tools to verify if locking is performed correctly: shared objects must be locked before use, unshared objects can only be accessed by the creating thread. Moreover, the verification can be done modularly, hence verification scales to large programs.
- Our programming model ensures absence of data races and deadlocks, and provides a sound approach for local reasoning about program behavior.
- We have prototyped a verifier as a custom build of the Spec# programming system [1, 2], and in particular its program verifier for sequential programs.
- Through a case study we show the model is usable in practice, and the annotation overhead is acceptable.

Our programming model builds on and extends the Spec# programming methodology [3] that enables sound reasoning about object invariants.
The rest of the paper is structured as follows. We introduce the methodology in three steps. The model of Section 2 prevents low-level data races on individual fields. Section 3 adds deadlock prevention. The final model, which adds prevention of races on data structures consisting of multiple objects, is presented in Section 4. Each section consists of three subsections, that elaborate the programming model, the program annotations, and the static verification rules, respectively. The remaining sections discuss additional features, experience, and related work, and offer a conclusion.

2 Preventing data races

A data race occurs when multiple threads simultaneously access the same variable, and at least one of these accesses is a write access. Developers can protect data structures accessed concurrently by multiple threads by associating a mutual exclusion lock with each data structure and ensuring that a thread accesses the data structure only when it holds the associated lock. However, mainstream programming languages such as Java and C# do not force threads to acquire any locks before accessing data structures, and they do not enforce that locks are associated with data structures consistently.
A simple strategy to prevent data races is to lock every object before accessing it. Although this approach is safe, it is rarely used in practice since it incurs a major performance penalty, is verbose, and is prone to deadlocks. Instead, standard practice is to only lock the objects that are effectively shared between multiple threads. However, it's hard to

distinguish shared objects (which should be locked) from unshared objects based on the program text. As a consequence, a compiler cannot enforce a locking discipline where shared objects can only be accessed when locked without additional annotations.
An additional complication is the fact that the implementation of a method may assume that an object is already locked by its caller. Hence, the implementation will access fields of a shared object without locking the object first. In such a case, merely indicating which objects are shared does not suffice. The implementor of a method should also make his assumptions about locks that are already held by the calling thread explicit in a method contract.
In this section, we describe a simple version of our programming model that deals with data races on the fields of shared objects. Later sections develop this model further to deal with deadlocks and high-level races on multi-object data structures.

2.1 Programming model

We describe our programming model in the context of Java, but it applies equally to C# and other similar languages.
In our programming model, accesses to shared objects are synchronized using Java's **synchronized** statement. A thread may enter a **synchronized** (o) block only if no other thread is executing inside a **synchronized** (o) block; otherwise, the thread waits. In the remainder of the paper, we use the following terminology to refer to Java's built-in synchronization mechanism: when a thread enters a **synchronized** (o) block, we say it *acquires o's lock* or, as a shorthand, that it *locks o*; while it is inside the block, we say it *holds o's lock*; and when it exits the block, we say it *releases o's lock*, or, as a shorthand, that it *unlocks o*. Note that, contrary to what the terminology may suggest, when a thread locks an object, the Java language prevents other threads from locking the object but it does not prevent other threads from accessing the object's fields. This is the main problem addressed by the proposed methodology. While a thread holds an object's lock, we also say that the object *is locked* by the thread.
An important terminological point is the following: when a thread t's program counter reaches a **synchronized** (o) block, we say the thread *attempts to lock o*. Some time may pass before the thread *locks o*, specifically if another thread holds o's lock. Indeed, if the other thread never unlocks o, t never locks o. The distinction is important because our programming model imposes restrictions on attempting to lock an object.
Our programming model prevents data races by ensuring that no two threads have access to a given object at any one time. Specifically, it conceptually associates with each thread t an *access set* $t.A$, which is the set of objects whose fields thread t is allowed to read or write at a given point, and the model ensures that no two threads' access sets ever intersect. Access sets can grow and shrink when objects are created, objects are shared, threads are created, or when a thread enters or exits a **synchronized** block. Note that these access sets do not exist at run

time: we use them to explain the programming model, and to implement the static verification.

- **Object creation.** When a thread creates a new object, the object is added to the creating thread's access set. This means the constructor can initialize the object's fields without acquiring a lock first. This also means single-threaded programs just work: if there is only a single thread, it creates all objects, and can access them without locking.
- **Object sharing.** In our model, the program state is extended with a new state variable for each object, called the object's *sharing mode*. This variable has two possible values: *unshared* and *shared*. Sharing modes, like access sets, are conceptual: they are not present at run time, but used to explain the model and implement the verification. A new object is initially in the unshared state. Threads other than the creating thread are not allowed to access its fields. In addition, no thread is allowed to attempt to lock an object in the unshared state: our programming model does not allow a **synchronized**(o){...} operation unless o is shared. In our programming model, objects that are not intended to be shared are never locked.
  If, at some point in the code, the developer wants to make the object available for concurrent access, he has to indicate this through an annotation (the **share** o annotation). From that point on, the object o is in the shared state, and threads can attempt to acquire the object's lock. When transitioning from the unshared to the shared state, the object is removed from the creating thread's access set. If, subsequent to this transition, any thread, including the creating thread, wishes to access the object, it must acquire its lock first.
  Once shared, an object can never revert to the unshared state.
- **Thread creation.** Starting a new thread transfers the accessibility of the receiver object of the thread's main method (i.e. the *Runnable* object in Java, or the *ThreadStart* delegate instance's target object in the .NET Framework) from the starting thread to the started thread. Otherwise, the thread's main method would not be allowed to access its receiver.
- **Acquiring and releasing locks.** When an object transitions to the shared state, it is removed from the creating thread's access set. Since the object is now not part of any thread's access set, no thread is allowed to access it. To gain access to such a shared object, a thread must lock the object first. When a thread acquires an object's lock, the object is added to that thread's access set. And vice versa, when a thread releases the lock, the object is removed from its access set.

As illustrated in Figure 1, an object can be in one of three states: *unshared*, *free* (not locked by any thread and shared) or *locked* (locked by some thread and shared). Initially, an object is unshared. Some objects will eventually transition to the shared state (at a program point indicated by the developer). After this transition, the object is not part of any thread's access set and is said to be *free*. To access a free object, it must be locked first, changing its state to locked and adding the object to the locking thread's access set. Unlocking the object removes it from the access set and makes it free again.

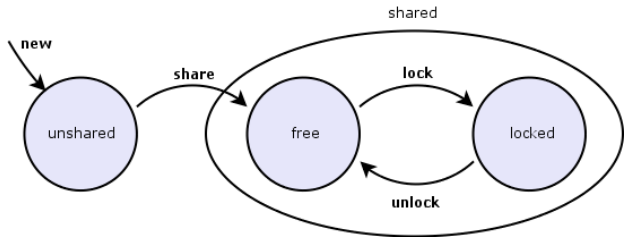

Fig. 1. The three states of an object.

Let's summarize. Threads are only allowed to access objects in their corresponding access set. A thread's access set consists of all objects whose lock it holds, and the objects it has created but not shared yet. Our programming model prevents data races by ensuring that access sets never intersect.

2.2 Program annotations

In this section we elaborate on the annotations needed by our approach by means of the example shown in Figure 2. The example consists of a program that observes events from different sources and keeps a count of the total number of events observed. Since the count is updated by multiple threads, it is subject to data races unless precautionary measures are taken. Our approach ensures that it's impossible to "forget" to take such measures.

In our prototype implementation (see Section 6), annotations are written as stylized comments. But to improve readability, we use a language integrated syntax in this paper.

The program shown in Figure 2 is a Java program augmented with a number of annotations (indicated by the gray background). More specifically, three sorts of annotations are used: **share** commands, **shared** modifiers and method contracts.

- The **share** command makes an unshared object available for concurrent access by multiple threads. In the example, the *counter* object is shared between all sessions.
- Fields and parameters can be annotated with a **shared** modifier, indicating they can only hold shared objects. The field *counter* of *Session* is an example of a field with a **shared** modifier.
- Method contracts are needed to make modular verification possible. They consist of preconditions and postconditions. A precondition states what the method implementation assumes about the current thread's access set (denoted as **tid**.A) and about the states of relevant objects. For instance, the precondition of the *run* method requires the receiver to be part of the current thread's access set. Postconditions state properties of access sets and object states which must hold when the method returns. For example, the postcondition of *Session*'s constructor guarantees that the new object is in the current thread's access set and that the new object is still unshared.

```
class Counter {
  int count;
  Counter()
    ensures this ∈ tid.A ∧ this.sharingMode = unshared;
  {}
}
class Session implements Runnable {
  shared Counter counter;
  int sourceId;
  Session(Counter counter, int sourceId)
    requires counter.sharingMode = shared;
    ensures this ∈ tid.A ∧ this.sharingMode = unshared;
  {
    this.counter := counter;
    this.sourceId := sourceId;
  }
  public void run()
    requires this ∈ tid.A;
  {
    for (;;) {
      // Wait for event from source sourceId (not shown)
      synchronized (counter) {
        counter.count++;
      }
    }
  }
}
class Program {
  static void start() {
    Counter counter := new Counter();
    share counter;
    new Thread(new Session(counter, 1)).start();
    new Thread(new Session(counter, 2)).start();
  }
}
```

Fig. 2. Example program illustrating the approach of Section 2.

Note that our annotations are entirely erasable, i.e. they have no effect whatsoever on the execution of the program.

The example program is correctly synchronized, and the annotations enable our static verifier to prove this. We discuss in the next subsection how this is done. If the developer forgets to write the **synchronized** block in the *run* method, the program is no longer correctly synchronized. Specifically, the access of *counter.count* in method *run* violates the programming model, since object *counter* is not in the thread's access set.

Thread creation To verify the example, we also need the method contracts of all library methods used by the program. These are shown in Figure 3.

The method contracts shown in Figure 3 encode the programming model's rules regarding thread creation.

- The *Thread* constructor requires its *Runnable* argument to be in the calling thread's access set. The constructor removes the *Runnable* object from the access set and associates it with the *Thread* object. Indeed, the constructor's postcondition does not state that in the post-state, the *Runnable* object is still in the access set, and therefore the caller cannot assume this and can no longer access the *Runnable* object.
- When method *start* is called, a new thread is started and the *Runnable* object associated with the *Thread* object is inserted into the new thread's access set. Method *run*'s precondition allows the method to assume that its receiver is in the access set.

```
public interface Runnable {
  void run();
    requires this ∈ tid.A;
}
public class Thread {
  public Thread(Runnable runnable)
    requires runnable ∈ tid.A;
    ensures this ∈ tid.A ∧ this.sharingMode = unshared;
  { ... }
  public void start()
    requires this ∈ tid.A;
  { ... }
}
```

Fig. 3. Contracts for the library methods used by the program in Figure 2.

2.3 Static verification

We have explained our programming model informally in the previous sections. In this section we define the model formally, and show how we can statically verify adherence to the model in a modular (i.e. per-method) way.

We proceed as follows: a program P enriched with our annotations is translated to a *verification-time* program P' enriched with assertions and classical method contracts. This translation defines the semantics of our annotations, and is the formal definition of our programming model: the original annotated program P is correct according to our model, if and only if the translated program P' is correct with respect to its assertions and classical method contracts. To check if the translated program P' is correct, we use an existing automatic program verifier for single-threaded programs. Our experiments show (Section 6) that state-of-the-art verifiers are capable of verifying realistic programs in this way.

The contributions of this paper are in the design of the annotation syntax (for the multithreading-specific annotations) and the translation of the annotated program; we use existing technology [2] for sequential program verification. The translation involves two things. In a first step, we insert additional verification-only fields and variables into the program (so called ghost fields and ghost variables) to track the state necessary to do the verification. The ghost variable **tid**.A represents the current thread's access set, and each object o is extended with a ghost field $o.sharingMode$, whose possible values are unshared and shared. We denote sets of objects as arrays of booleans indexed by objects. For example, $S[o] \leftarrow true$ adds object o to set S.

Then, in a second step each method of the original program is translated in such a way that the translated method can be verified modularly. The method contracts that the developer writes in annotations are classical method contracts on the ghost state introduced in the first step. The code and other annotations written by the developer are translated into verification-time code and proof obligations (written as assertions) for the verifier. The essence of the translation of code and annotations is shown in Figure 4. It is a formalization of the programming model rules introduced in Section 2.1. We ignore the fact that object references can be null to reduce clutter. The verification-time code for a **synchronized** block includes a **havoc** operation that assigns an arbitrary value to all fields of the object being locked. This reflects the fact that other threads may have modified these fields. Source program assignment and verification-time assignment are shown as $:=$ and $\leftarrow$, respectively.

3 Lock levels for deadlock prevention

The approach of Section 2 prevents data races but it does not prevent deadlocks. In this section, we introduce our approach to deadlock prevention.

For the purpose of this paper, we define a deadlock to be a cycle of threads such that each thread is waiting for the next thread to release

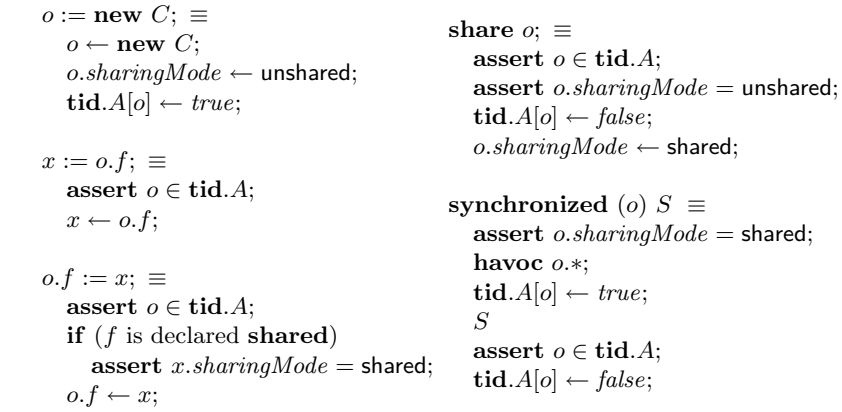

Fig. 4. Translation of source program commands to verification-time commands.

some lock. Formally, a deadlock is a sequence of threads $t_0, \ldots, t_{n-1}$ and a sequence of objects $o_0, \ldots, o_{n-1}$ such that t_i holds o_i's lock and is trying to acquire $o_{(i+1) \bmod n}$'s lock. Threads involved in a deadlock are stuck forever.

The prototypical way in which a developer can avoid deadlocks is by defining a partial order over all shared objects, and by allowing a thread to attempt to acquire an object's lock only if the object is less than all objects whose lock the thread already holds.

There are different common strategies for defining such a partial order. A first one is to define the order statically. This approach is common in case the shared objects protect global resources: code will have to acquire these resources in the statically defined order. A second strategy is to define the order based on some field of the objects involved. For instance to define a transfer operation between accounts, the two accounts involved can be locked in order of the account number, thus avoiding deadlocks while locking account objects.

In some cases the developer of a particular module may only wish to impose partial constraints on the locking order or may wish to abstract over a set of objects. For instance the developer of the Subject class in the Subject-Observer pattern may wish to specify that Observers should be locked before locking the Subject and not vice-versa. In other words, all Observers are above the Subject in the deadlock prevention ordering.

3.1 Programming model

Our programming model is designed to support all three scenarios outlined above. The developer can indicate his intended ordering through the intermediary of *lock levels*. A lock level is a value of the new primitive type (existing only for verification purposes) **locklevel**. A new lock level can be constructed between given existing lock levels using the constructor **between**$(\{\ell_1^A, \ldots, \ell_m^A\}, \{\ell_1^B, \ldots, \ell_n^B\})$, where $0 \leq m, n$, provided that

each specified lower bound is below each specified upper bound; formally, for each $1 \leq i \leq m$ and $1 \leq j \leq n$, $\ell_i^A < \ell_j^B$. The new value is above $\ell_1^A, \ldots, \ell_m^A$ and below $\ell_1^B, \ldots, \ell_n^B$. There is no other way to construct a lock level, which ensures that the less-than ($<$) relation on lock levels is always a partial order.
In the model, a lock level is associated with an object the moment the object is shared. This defines the lock order: for shared objects o_1 and o_2, we have $o_1 < o_2$ iff $o_1.\mathit{lockLevel} < o_2.\mathit{lockLevel}$. A thread is only allowed to lock an object if the object is less than the objects whose lock the thread already holds.
The level of indirection introduced by the lock levels provides an easy way to abstract over sets of objects. In the Subject-Observer example discussed above, all Observer objects can be given the same lock level (that should be above the Subject lock level).

3.2 Program annotations

In a concurrent Java or C# program, a lock ordering adopted by the developers of a program for the purpose of deadlock prevention is not explicit in the program text, although it can be documented informally in comments. We propose annotations that make it possible for a developer to document the intended ordering formally. As a consequence, static verification of adherence to the ordering is possible (Section 3.3).
Three kinds of annotations are important. We discuss them using the example of the Dining Philosophers program in Figure 5. The program implements a deadlock-free solution to the Dining Philosophers problem with three philosophers. Our annotations explain formally why the program is deadlock-free.
The first kind of annotation is the creation of a lock level using the **between** constructor. The example defines the lock levels and their ordering statically in class *Program*'s *start* method. Three linearly ordered levels are defined: *level1* $<$ *level2* $<$ *level3*.
The second kind of annotation associates lock levels with shared objects. The **share** annotation is extended to accept a lock level as the second argument. Again, this happens three times in the example: each of the forks is shared with its associated lock level. As a consequence, fork objects are totally ordered, with *fork1* $<$ *fork2* $<$ *fork3*. Hence, forks can only be locked in descending order.
The third kind of annotations are the method contracts that make modular static verification possible. Method contracts make explicit what assumptions the method makes about the ordering of parameter objects, or about locks already held by the current thread. For instance the constructor of *Philosopher* expects its first argument to have a lower lock level than the second argument, and the *run* method requires that the current thread holds no locks.
These annotations enable a formal static verification of deadlock-freeness.

3.3 Static verification

Static verification is again done by translating the annotated program P into a program P' enriched with proof obligations for a static verifier (in

```
class Fork {
}
class Philosopher implements Runnable {
  shared Fork fork1;
  shared Fork fork2;

  Philosopher( shared Fork fork1, shared Fork fork2)
    requires fork1.lockLevel < fork2.lockLevel;
    ensures this ∈ tid.A ∧ this.sharingMode = unshared;
  {
    this.fork1 := fork1;
    this.fork2 := fork2;
  }
  public void run()
    requires this ∈ tid.A;
    requires tid.lockStack.isEmpty();
  {
    for (;;) {
      synchronized (fork2) {
        synchronized (fork1) {
          // Use the forks to eat...
        }
      }
    }
  }
}
class Program {
  static void start() {
    locklevel level1 := between({}, {});
    locklevel level2 := between({level1}, {});
    locklevel level3 := between({level2}, {});
    Fork fork1 := new Fork();
    share (fork1, level1);
    Fork fork2 := new Fork();
    share (fork2, level2);
    Fork fork3 := new Fork();
    share (fork3, level3);
    new Thread(new Philosopher(fork1, fork2)).start();
    new Thread(new Philosopher(fork2, fork3)).start();
    new Thread(new Philosopher(fork1, fork3)).start();
  }
}
```

Fig. 5. Deadlock prevention for the Dining Philosophers

the form of classical method contracts and assertions). The translation adds ghost fields and variables to track the necessary state. To track the lock level of objects, we add to each object a ghost field called *lockLevel*, whose value is either *null* or a lock level and whose initial value is *null*. The field is written only once: when the object is shared a non-null lock level is assigned to this field. This way, each shared object has an immutable association with a lock level.
To track the locks that the current thread holds, we introduce a ghost variable **tid**.*lockStack*, which is a stack containing the objects whose lock the thread holds. Whenever a thread acquires an object's lock, the object is pushed onto the stack. Note that it follows that the top of the stack is always the least of all objects on the stack. A thread is allowed to acquire an object o's lock only if the lock stack is empty or o's lock level is strictly less than the lock level of the object at the top of the stack. We denote this condition as $o < \mathbf{tid}.lockStack$.
The essence of the translation of an annotated program is summarized in Figure 6.

$o := \mathbf{new}\ C;\ \equiv$
  $o \leftarrow \mathbf{new}\ C;$
  $o.sharingMode \leftarrow \mathsf{unshared};$
  $\mathbf{tid}.A[o] \leftarrow true;$

$x := o.f;\ \equiv$
  assert $o \in \mathbf{tid}.A;$
  $x \leftarrow o.f;$

$o.f := x;\ \equiv$
  assert $o \in \mathbf{tid}.A;$
  if (f is declared **shared**)
    assert $x.sharingMode = \mathsf{shared};$
  $o.f \leftarrow x;$

share $(o, l);\ \equiv$
  assert $o \in \mathbf{tid}.A;$
  assert $o.sharingMode = \mathsf{unshared};$
  $\mathbf{tid}.A[o] \leftarrow false;$
  $o.sharingMode \leftarrow \mathsf{shared};$
  $o.lockLevel \leftarrow l;$

synchronized $(o)\ S\ \equiv$
  assert $o.sharingMode = \mathsf{shared};$
  assert $o < \mathbf{tid}.lockStack;$
  $\mathbf{tid}.lockStack.push(o);$
  havoc $o.*;$
  $\mathbf{tid}.A[o] \leftarrow true;$
  S
  assert $o \in \mathbf{tid}.A;$
  $\mathbf{tid}.A[o] \leftarrow false;$
  $\mathbf{tid}.lockStack.pop();$

Fig. 6. Translation of source program commands to verification-time commands.

4 Invariants and Ownership

The approach as described in the preceding sections ensures absence of low-level data races and deadlocks. However, it does not prevent higher-level race conditions, where the programmer protects individual field accesses, but not updates involving accesses of multiple fields or objects that are part of the same data structure. As a result, accesses may be interleaved in such a way that the data structure's consistency is not maintained.

4.1 Programming model

To prevent race conditions that break the consistency of multi-object data structures, we integrate the Spec# methodology's object invariant and ownership system [3] into our approach, to obtain the final programming model of this paper. This model supports objects that use other objects to represent their state, and object invariants that express consistency constraints on such multi-object structures.

The programming model requires the programmer to designate a subset of each class's fields as the class's *rep fields*. The objects pointed to by an object o's non-null *rep* fields in a given program state are called o's *rep objects*. An object's *rep* objects may have *rep* objects themselves, and so on; we refer to all of these as the object's transitive *rep* objects. The fields of an object, along with those of its transitive *rep* objects, are considered in our approach to constitute the entire representation of the state of the object; hence the name. As will be explained later, a shared object o's lock protects both o and its transitive *rep* objects.

In addition to a set of *rep* fields, the programming model requires the programmer to designate, for each class C, an *object invariant*, denoted $Inv_C(o)$ when applied to an object o of C. $Inv_C(o)$ is a predicate that may depend on the state of o, i.e. the fields of o and of its transitive *rep* objects.

The object invariant for an object o need not hold in each program state; rather, the programming model associates with each object a boolean state variable called its *inv bit*.[1] The programming model requires the object invariant to hold only when the *inv* bit is *true*.

The programming model requires an object's *inv* bit to be *true* when a thread shares the object or unlocks it, i.e. when the object becomes free. It follows that each free object's *inv* bit is *true* and its object invariant holds. As a result, when a thread locks an object, it may assume that the object's *inv* bit is true and its object invariant holds.

At the start of an object's constructor, its *inv* bit is *false*. The programming model requires the programmer to designate the regions of code where an object's invariant is supposed to hold by designating the points where **pack** o; and **unpack** o; operations occur. The former sets o's *inv* bit to *true*, and the latter sets it to *false*.

To ensure that whenever an object's *inv* bit is *true*, its object invariant holds, the programming model imposes the following restrictions:

- A thread may assign to an object's fields only when the object is in the thread's access set *and* the object's *inv* bit is *false*. Furthermore, the remaining restrictions ensure that whenever an object's *inv* bit is *true*, then so are those of its transitive *rep* objects. As a result, an object's state does not change while its *inv* bit is *true*.
- A thread is allowed to perform a **pack** o; operation only when o's object invariant holds, its *inv* bit is *false*, and the *inv* bits of o's *rep* objects are *true*. Furthermore, besides setting o's *inv* bit to *true*, the operation removes o's *rep* objects from the thread's access set.

[1] Like the sharing mode, the *inv* bit is not a field in the actual program; it is a state variable introduced only to explain the programming model.

– A thread is allowed to perform an **unpack** o; operation only when o's *inv* bit is *true*. The operation sets o's *inv* bit to *false* and adds o's *rep* objects to the thread's access set.

We say that an object *owns* its *rep* objects whenever its *inv* bit is *true*. It follows from the above restrictions that an object has at most one owner. Note that our approach supports ownership transfer; a *rep* object can be moved from one owner to another by first unpacking both owners and then simply updating the relevant *rep* fields.

4.2 Program annotations

The example in Figure 7 shows the annotations required by our final methodology. A *Rectangle* object is used to store the bounds of an application's window. The *Rectangle*'s state is represented internally using two *Point* objects, that represent the location of upper-left and lower-right corner, respectively. If the user drags the window's title bar, the window manager moves the window, even if the application is painting the window contents. Our methodology ensures that the application sees only valid states of the *Rectangle* object.
Developers designate a class's *rep* fields using the **rep** modifier, they define a class's object invariant using **invariant** declarations, and they insert **pack** and **unpack** commands in method bodies. Additionally, developers may denote an object o's *inv* bit in method contracts, using the $o.inv$ notation.

4.3 Static verification

Figure 8 shows the translation of source program commands to input for the sequential program verifier.
Note that the verification-time commands for a **synchronized** (o) block havoc all objects that are not in the thread's access set, rather than just object o. This is necessary since other threads may have modified not just o, but o's transitively owned objects as well. Also, the assumption encoded by the **assume** statement is justified by the programming model, as explained above.
The verifier is additionally made aware of the following properties:

$$(\forall o \bullet o.inv \Rightarrow Inv(o))$$
$$(\forall o, p \bullet o.inv \wedge p \in \text{repobjects}(o) \Rightarrow p.inv)$$

These are guaranteed to hold in each program state by the programming model, as explained above.

5 Additional features

In this section we briefly describe how our approach supports subclassing, and how it supports sharing immutable objects without synchronization.

```
class Point {
  int x, y;
  void move(int dx, int dy)
    requires this ∈ tid.A ∧ this.inv; ensures this ∈ tid.A ∧ this.inv;
  { unpack this; x := x + dx; y := y + dy; pack this; }
}
class Rectangle {
  rep Point ul, lr;
  invariant ul.x ≤ lr.x ∧ ul.y ≤ lr.y;
  void move(int dx, int dy)
    requires this ∈ tid.A ∧ this.inv; ensures this ∈ tid.A ∧ this.inv;
  { unpack this; ul.move(dx, dy); lr.move(dx, dy); pack this; }
  int getHeight()
    requires this ∈ tid.A ∧ this.inv; ensures this ∈ tid.A ∧ this.inv;
    ensures 0 ≤ result;
  { unpack this; int h := lr.y − ul.y; pack this; return h; }
}
class Application {
  shared Rectangle windowBounds;
  void paint()
    requires tid.lockStack.isEmpty();
    requires this ∈ tid.A ∧ this.inv; ensures this ∈ tid.A ∧ this.inv;
  {
    int height;
    synchronized (windowBounds) {
      height := windowBounds.getHeight();
    }
    ...
  }
}
class WindowManager {
  shared Rectangle windowBounds;
  void mouseDragged(int dx, int dy)
    requires tid.lockStack.isEmpty();
    requires this ∈ tid.A ∧ this.inv; ensures this ∈ tid.A ∧ this.inv;
  {
    synchronized (windowBounds) {
      windowBounds.move(dx, dy);
    }
  }
}
```

Fig. 7. An example illustrating our data race and deadlock prevention strategy, combined with object invariants and ownership.

```
o := new C; ≡
  o ← new C;
  o.sharingMode ← unshared;
  tid.A[o] ← true;
  o.inv ← false;

pack o; ≡
  assert o ∈ tid.A;
  assert ¬o.inv
  assert (∀p ∈ repobjects(o) •
    p ∈ tid.A ∧ p.inv);
  assert Inv(o);
  o.inv ← true;
  foreach (p ∈ repobjects(o))
    tid.A[p] ← false;

unpack o; ≡
  assert o ∈ tid.A;
  assert o.inv;
  o.inv ← false;
  foreach (p ∈ repobjects(o))
    tid.A[p] ← true;

x := o.f; ≡
  assert o ∈ tid.A;
  x ← o.f;

o.f := x; ≡
  assert o ∈ tid.A;
  assert ¬o.inv;
  if (f is declared shared)
    assert x.sharingMode = shared;
  o.f ← x;

share (o, l); ≡
  assert o ∈ tid.A;
  assert o.inv;
  assert o.sharingMode = unshared;
  o.lockLevel ← l;
  o.sharingMode ← shared;
  tid.A[o] ← false;

synchronized (o) S ≡
  assert o.sharingMode = shared;
  assert o < tid.lockStack;
  tid.lockStack.push(o);
  foreach (p ∉ tid.A) havoc p.∗;
  tid.A[o] ← true;
  assume o.inv;
  S
  assert o ∈ tid.A;
  assert o.inv;
  tid.A[o] ← false;
  tid.lockStack.pop();
```

Fig. 8. Translation of source program commands to verification-time commands (with invariants and ownership).

5.1 Subclassing

All fields of a shared object are protected by the object's lock, even if those fields are not all declared by the same class.

Each class may declare an object invariant, so if an object is an instance of multiple classes, multiple object invariants apply to it. We adopt the Spec# methodology's approach [3] by allowing an object to be *fully packed*, which means all object invariants need to hold, *fully unpacked*, which means none need to hold, or *partially packed down to class C*, which means all object invariants declared by class C and those declared by its direct and indirect superclasses need to hold.

Our approach enforces the property that objects that are shared but not locked (i.e., objects that are free) are fully packed.

5.2 Immutable objects

If after an object is shared, it is only ever inspected and never mutated, then there's no need to synchronize accesses. Our approach supports

this by splitting a thread's access set into a *read set* and a *write set*, and by splitting the shared sharing mode into a lockprotected mode and an immutable mode. Correspondingly, the **share** command is replaced with a **share_lockprotected** command and a **share_immutable** command. Sharing an object as immutable requires that it is unshared and in the current thread's write set. It removes the object from the write set and adds it to each thread's read set (even if the thread has not yet been started). If the object has *rep* objects, they are recursively shared as immutable and added to all read sets.
Whether an object is shared as lock-protected or as immutable, it must be fully packed in both cases. As a result, an immutable object's invariant holds at all times.
Our approach supports writing classes that allow client code the freedom to use some of the class's objects as thread-local (unshared) objects, to share some and protect them by their lock, and to share some as immutable. Such a class typically provides inspector methods and mutator methods. Only inspector methods can be called on immutable objects.
The **unpack** *o*; command requires *o* to be in the thread's write set. To allow an inspector method to access its receiver's *rep* objects, regardless of whether the receiver is writable or only readable, our approach includes a **read** (*o*) block that adds *o*'s *rep* objects to the thread's read set for the duration of the block. It also temporarily removes *o* itself from the write set (but not the read set); this is required for soundness.

6 Experience

To verify the applicability of our approach to realistic, useful programs, we implemented it in a custom build of the Spec# program verifier [2] and used it to verify a chat server application written in C# with annotations inserted in the form of specially marked comments. The application verifies successfully; this guarantees the following:

- The program is free from data races and deadlocks
- Object invariants, loop invariants, method preconditions and post-conditions, and assert statements declared by the program hold
- The program is free from null dereferences, array index out of bounds errors, and typecasting errors
- The program is free from races on platform resources such as network sockets. This is achieved by enforcing concurrency contracts on the relevant API methods.

Table 1 shows the annotation overhead of four programs which we annotated and verified. Programs chat and phone were derived from the ones used in [4].
The prototype verifier and the sample programs are available from the first author's web site at http://www.cs.kuleuven.be/~bartj/.

7 Related Work

The Extended Static Checkers for Modula-3 [5] and for Java [6] attempt to statically find errors in object-oriented programs. These tools include

Program	Lines of Code	Lines Changed or Added	Overhead
chat	344	117	34%
phone	222	50	23%
prod-cons	84	24	29%
philosophers	64	21	33%

Table 1. Annotation overhead

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 [7, 8] 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.* [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 (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.

Boyapati *et al.* prevent deadlocks by allowing the developer to declare a fixed set of lock levels. Lock levels are assigned to objects as type arguments. Additional expressiveness is gained by supporting locking the nodes of a mutable tree data structure or an immutable DAG data structure, and by ordering the objects of designated classes at run time.

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], by way of a type system [10], or using a theorem prover [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 Figure 8). Also, in [10] 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 [10] 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.

Ábrahám-Mumm *et al.* [12] 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 [13].

There are also tools that try dynamically to detect violations of safe concurrency. A notable example is Eraser [14]. 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.

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 [15].

The present approach evolved from the approach originally published in [16]. It improves upon it by directly supporting platform-standard locking primitives, by preventing deadlocks, by adding support for immutable objects, and by reporting on experience gained using a prototype implementation.

8 Conclusion

We propose a programming model for concurrent programming in Java-like languages, and the design of a set of program annotations that make the use of the programming model explicit and that enable automated verification of compliance. Our programming model ensures absence of data races and deadlocks, and provides a sound approach for local reasoning about program behavior. We have prototyped the verifier as a custom build of the Spec# programming system. Through a case study we show the model is usable in practice, and the annotation overhead is acceptable.

Our verification approach is sound; the proof of soundness is largely analogous to the one given in [17] for an earlier version of the approach. We are currently further extending the programming model to encompass static fields, lock re-entry, and read-write locks.

References

1. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: CASSIS. Volume 3362 of LNCS., Springer (2004)

2. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: Proceedings of the Fourth International Symposium on Formal Methods for Components and Objects (FMCO 2005). (2006) To appear.
3. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology **3**(6) (2004) 27–56
4. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: Preventing data races and deadlocks. In: OOPSLA 2002. Volume 37 of SIGPLAN Notices., ACM (2002) 211–230
5. Detlefs, D.L., Leino, K.R.M., Nelson, G., Saxe, J.B.: Extended static checking. Research Report 159, Compaq Systems Research Center (1998)
6. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI 2002. Volume 37 of SIGPLAN Notices., ACM (2002) 234–245
7. Qadeer, S., Rajamani, S.K., Rehof, J.: Summarizing procedures in concurrent programs. In: POPL 2004. Volume 39 of SIGPLAN Notices., ACM (2004) 245–255
8. Freund, S.N., Qadeer, S.: Checking concise specifications for multithreaded software. Journal of Object Technology **3**(6) (2004) 81–101
9. Flanagan, C., Freund, S.N.: Atomizer: A dynamic atomicity checker for multithreaded programs. In: POPL 2004. Volume 39 of SIGPLAN Notices., ACM (2004) 256–267
10. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: PLDI 2003, ACM (2003) 338–349
11. Rodríguez, E., Dwyer, M., Flanagan, C., Hatcliff, J., Leavens, G.T., Robby: Extending sequential specification techniques for modular specification and verification of multi-threaded programs. In: ECOOP 2005. Volume 3586 of LNCS., Springer (2005) 551–576
12. Ábrahám-Mumm, E., de Boer, F.S., de Roever, W.P., Steffen, M.: Verification for Java's reentrant multithreading concept. In: FoSSaCS 2002. Volume 2303 of LNCS., Springer (2002) 5–20
13. Hoare, C.A.R.: Monitors: An operating system structuring concept. Communications of the ACM **17**(10) (1974) 549–557
14. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.E.: Eraser: A dynamic data race detector for multi-threaded programs. ACM Transactions on Computer Systems **15**(4) (1997) 391–411
15. Welc, A., Jagannathan, S., Hosking, A.L.: Transactional monitors for concurrent objects. In: ECOOP 2004. Volume 3086 of LNCS., Springer (2004)
16. Jacobs, B., Leino, K.R.M., Piessens, F., Schulte, W.: Safe concurrency for aggregate objects with invariants. In: Proc. Int. Conf. Software Engineering and Formal Methods (SEFM 2005), IEEE Computer Society (2005) 137–146
17. Jacobs, B., Leino, K.R.M., Piessens, F., Schulte, W.: Safe concurrency for aggregate objects with invariants: Soundness proof. Technical Report MSR-TR-2005-85, Microsoft Research (2005)