Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 174 (2007) 23–47
www.elsevier.com/locate/entcs

A Simple Sequential Reasoning Approach for Sound Modular Verification of Mainstream Multithreaded Programs

Bart Jacobs, Jan Smans, Frank Piessens[1,2]

DistriNet, Dept. Computer Science
Katholieke Universiteit Leuven
Leuven, Belgium

Wolfram Schulte[3]

Microsoft Research
Redmond, WA, USA

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 multi-threading 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. Our approach supports immutable objects as well as static fields and static initializers. 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.

Keywords: Aliasing, class initialization, concurrency, data races, deadlocks, immutable objects, local reasoning, modular reasoning, ownership, verification condition generation

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.

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

[2] Email: {bartj,jans,frank}@cs.kuleuven.be

[3] Email: schulte@microsoft.com

doi:10.1016/j.entcs.2007.04.005

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 consequence it is practically impossible for the compiler or other static analysis tools to verify if locking is performed correctly.

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.

Several other approaches exists to verify race- and deadlock-freedom for multithreaded code. They range from generating verification conditions [6,8,10,17,1,18], to type systems [5,9]. (See Section 8 for an overview of related work.)

Our approach is unique, in that it builds around protecting invariants and that it allows sequential reasoning for multithreaded code. The contributions of this paper are thus as follows:

- We present a programming model and a set of annotations for concurrent programming in Java-like languages.
- Following our programming model ensures absence of data races and deadlocks.
- The generated verification conditions allow sound local reasoning about program behavior. Note that in this paper we ignore null dereference checking to avoid clutter, although our prototype fully supports it.
- We have prototyped a verifier as a custom build of the Spec# programming system [4,2], 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.

The present approach evolved from [12] and [14]. It improves upon [12] 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. It improves upon [14] by adding support for static fields and static initializers. As did [12] and [14], it builds on and extends the Spec# programming methodology [3] that enables sound reasoning about object invariants in sequential programs.

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 immutable objects, our approach for static fields and static initializers, 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 is 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 addition to an access set, our model associates with each run-time state a global *shared set* S. We call the objects in S *shared* and objects in the complement of S *unshared*. The shared set, like the access sets, is conceptual: it is not present at run time, but used to explain the model and implement the verification.

  A new object is initially unshared. Threads other than the creating thread are not allowed to access its fields. In addition, no thread is allowed to attempt to lock an unshared object: 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 shared, and threads can attempt to acquire the object's lock. When an object is being shared, the object is removed from the creating thread's access set and added to the shared 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.

  In addition, the precondition requires this receiver object to be unshared. As a consequence, the invariant that shared objects in a thread's access set are also locked by that thread is maintained.

- **Acquiring and releasing locks.** When an object is being shared, it is removed from the creating thread's access set and added to the shared 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 for the duration of the synchronized block.

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.

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, the objects it has created but not shared yet, and of the receiver object of the thread's main method, if the thread did not share this object 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 is impossible to "forget" to take such measures.

In our prototype implementation (see Section 7), 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. Furthermore, := denotes assignment and = equality.

- The **share** command makes an unshared object available for concurrent access

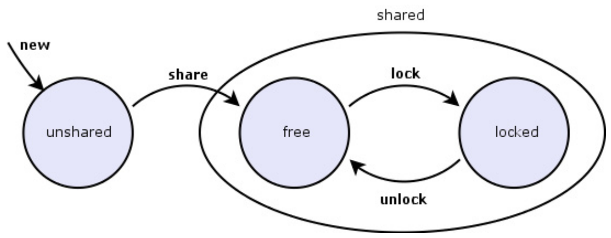

Fig. 1. The three states of an object.

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 global shared set. For instance, the precondition of the *start* method requires the access set to be empty. Postconditions state properties of access sets and the shared set. For example, the postcondition of *Session*'s constructor guarantees that the new object is in the current thread's access set and unshared.

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.

2.2.1 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 argument to be part of the calling thread's access set and unshared. 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

```
class Counter {
   int count;
   Counter()
      ensures this ∈ tid.A ∧ this ∉ S;
   {}
}
class Session implements Runnable {
   shared Counter counter;
   int sourceId;
   Session(Counter counter, int sourceId)
      requires counter ∈ S;
      ensures this ∈ tid.A ∧ this ∉ S;
   {
      this.counter := counter;
      this.sourceId := sourceId;
   }
   public void run()
      requires tid.A = {this} ∧ this ∉ S;
   {
      for (; ; ) {
         // Wait for event from source sourceId (not shown)
         synchronized (counter) {
            counter.count++;
         }
      }
   }
}
class Program {
   static void start()
      requires tid.A = ∅;
   {
      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. Programmer-supplied annotations are shown on a gray background.

```
public interface Runnable {
  void run();
    requires $\mathbf{tid}.A = \{\mathbf{this}\} \wedge \mathbf{this} \notin S$;
}
public class Thread {
  public Thread(Runnable runnable)
    requires $runnable \in \mathbf{tid}.A \wedge runnable \notin S$;
    ensures $\mathbf{this} \in \mathbf{tid}.A \wedge \mathbf{this} \notin S$;
  { ... }
  public void start()
    requires $\mathbf{this} \in \mathbf{tid}.A$;
  { ... }
}
```

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

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 the only object in the access set and that this object is unshared.

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 7) 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 two verification-only variables into the program (so called ghost variables) to track the state necessary to do the verification. The ghost variable $\mathbf{tid}.A$ represents the current thread's access set, while S represents the set of shared objects.

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. Additionally, it also includes a **havoc where** operation which replaces the shared set by an arbitrary superset. This reflects the fact that other threads may have modified these fields. Source program assignment and verification-time assignment are shown as := and ←, respectively.

```
$o :=$ new $C$; $\equiv$
    $o \leftarrow$ new $C$;
    assume $o \notin S$;
    tid.$A \leftarrow$ tid.$A \cup \{o\}$;

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

$o.f := x$; $\equiv$
    assert $o \in$ tid.$A$;
    if ($f$ is declared shared)
        assert $x \in S$;
    $o.f \leftarrow x$;
```

```
share $o$; $\equiv$
    assert $o \in$ tid.$A$;
    assert $o \notin S$;
    tid.$A \leftarrow$ tid.$A \setminus \{o\}$;
    tid.$S \leftarrow$ tid.$S \cup \{o\}$;

synchronized ($o$) $B$ $\equiv$
    assert $o \in S$;
    assert $o \notin A$;
    havoc $o.*$;
    tid.$A \leftarrow$ tid.$A \cup \{o\}$;
    $B$
    tid.$A \leftarrow$ tid.$A \setminus \{o\}$;
```

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

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 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

$$\textbf{between}(\{\ell_1^A, \ldots, \ell_m^A\}, \{\ell_1^B, \ldots, \ell_n^B\})$$

, 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-

```
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 ∉ S
  {
    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()
    requires tid.lockStack.isEmpty();
  {
    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

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 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.

The essence of the translation of an annotated program is summarized in Figure 6. Note that the rules for object creation and field access have been omitted since they are unchanged from the previous section.

```
share $(o, l);$ $\equiv$
    assert $o \in \mathbf{tid}.A$;
    assert $o \notin S$;
    $\mathbf{tid}.A \leftarrow \mathbf{tid}.A \setminus \{o\}$;
    $\mathbf{tid}.S \leftarrow \mathbf{tid}.S \cup \{o\}$;
    $o.\mathit{lockLevel} \leftarrow l$;
```

```
synchronized $(o)$ $B$ $\equiv$
    assert $o \in S$;
    assert $o < \mathbf{tid}.\mathit{lockStack}$;
    $\mathbf{tid}.\mathit{lockStack}.\mathit{push}(o)$;
    havoc $o.*$;
    $\mathbf{tid}.A \leftarrow \mathbf{tid}.A \cup \{o\}$;
    $B$
    $\mathbf{tid}.A \leftarrow \mathbf{tid}.A \setminus \{o\}$;
    $\mathbf{tid}.\mathit{lockStack}.\mathit{pop}()$;
```

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.*[4] 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:

[4] The *inv* bit is not a field in the actual program; it is a variable introduced only to explain the programming model.

- 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.
- 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 **synchro-nized** (*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:

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

```
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.

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

5 Immutable objects

In this section we briefly describe how the approach we implemented supports sharing immutable objects without synchronization.

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_im-mutable** 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.

```
$o :=$ new $C$; $\equiv$
    $o \leftarrow$ new $C$;
    assume $o \notin S$;
    tid.$A \leftarrow$ tid.$A \cup \{o\}$;
    $o.inv \leftarrow false$;

pack $o$; $\equiv$
    assert $o \in$ tid.$A$;
    assert $\neg o.inv$
    assert $(\forall p \in \text{repobjects}(o) \bullet$
        $p \in$ tid.$A \wedge p \notin S \wedge p.inv)$;
    assert $Inv(o)$;
    $o.inv \leftarrow true$;
    foreach $(p \in \text{repobjects}(o))$
        tid.$A \leftarrow$ tid.$A \setminus \{p\}$;

unpack $o$; $\equiv$
    assert $o \in$ tid.$A$;
    assert $o.inv$;
    $o.inv \leftarrow false$;
    foreach $(p \in \text{repobjects}(o))\{$
        tid.$A \leftarrow$ tid.$A \cup \{p\}$;
        assume $p \notin S$;
    $\}$

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

$o.f := x$; $\equiv$
    assert $o \in$ tid.$A$;
    assert $\neg o.inv$;
    if ($f$ is shared)
        assert $x \in S$;
    $o.f \leftarrow x$;

share $(o, l)$; $\equiv$
    assert $o \in$ tid.$A$;
    assert $o.inv$;
    assert $o \notin S$;
    $o.lockLevel \leftarrow l$;
    $S \leftarrow S \cup \{o\}$;
    tid.$A \leftarrow$ tid.$A \setminus \{o\}$;

synchronized $(o)$ $B$ $\equiv$
    assert $o \in S$;
    assert $o <$ tid.$lockStack$;
    tid.$lockStack.push(o)$;
    foreach $(p \notin$ tid.$A)$
        havoc $p.*$;
    tid.$A \leftarrow$ tid.$A \cup \{o\}$;
    assume $o.inv$;
    $B$
    assert $o.inv$;
    tid.$A \leftarrow$ tid.$A \setminus \{o\}$;
    tid.$lockStack.pop()$;
```

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

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 Static fields and static initializers

In this section, we extend our programming model to also prevent data races on static fields and deadlocks involving class initialization and class lock acquisition. The extended model also enforces invariants on the static fields of a class and its transitively owned objects.

We first briefly recall the syntax and semantics of class initialization in Java. We then present the programming model extension. The remaining subsections discuss acyclicity of the **lock_before** relation and describe our support for classes whose static state does not change after initialization.

6.1 Class initialization in Java

In this section, we briefly recall the syntax and semantics of class initialization in the Java programming language. In the next section, we explain our approach for modular verification of Java programs with static fields and static initializers.

A class may declare static field initializers and static initializer blocks (or *static initializers* for short). In the sequel, we assume that each class declares no static field initializers and exactly one static initializer. (It is always possible to rewrite a class to satisfy this assumption.) A static initializer is an arbitrary sequence of statements.

Java ensures that a class's static initializer is executed at most once, at the last possible moment, and that it completes normally before any access of the class or one of its transitive subclasses (except for an access performed while the current thread is running the static initializer). The following are considered accesses of a class: reads and writes of static fields, calls of static methods, and calls of constructors.

We may think of this semantics as follows. In a preprocessing step, an **initialize** C; command is inserted before each access of class C in the program.[5] An **initialize** C; command is additionally inserted at the start of the static initializer of each direct subclass of C. A thread t executes an **initialize** C; command as follows:

- If no thread has started executing C's static initializer, thread t executes C's static initializer. If execution completes normally, the **initialize** operation completes normally. If execution completes abruptly with an exception, then so does the **initialize** operation.
- If thread t is currently executing C's static initializer, i.e. if this is a recursive **initialize** C; operation, then the operation completes normally directly, without recursively executing the static initializer.
- Otherwise, if some other thread is executing C's static initializer, thread t waits until the execution has completed (normally or abruptly). Once execution of C's static initializer has completed normally or abruptly, execution of the

[5] Actually, for calls of constructors and static methods, we insert the **initialize** operation at the top of the constructor or method body rather than at the call site. Both encodings are sound but the callee-side encoding yields slightly simpler method contracts.

initialize C; operation by thread t completes normally or abruptly in the same way.

Note that class initialization may deadlock, if threads are waiting for each other to finish executing a static initializer.

6.2 Programming Model

In this section, we present a programming model that prevents data races on static fields and deadlocks involving class initialization and class lock acquisition, and that ensures invariants on the static fields of a class and its transitively owned objects.

Our general approach is to treat static fields of a class C as if they are fields of the *Class* object for C, denoted in Java as C.**class**. That is:

- We prevent data races on static fields by allowing a thread t to access a static field $C.f$ only if C.**class** is in t's access set, and by ensuring that access sets are always disjoint.
- An object C.**class** is accessible and unshared on entry to its static initializer. Upon normal completion of the static initializer, a **share** operation is implicitly performed on the object. Once C.**class** is shared, to access the static fields of C, a thread must first lock C.**class**, which adds C.**class** to the thread's access set.
- The lock acquisition deadlock prevention approach applies to *Class* objects as well. A class C may specify lower bounds for its *Class* object's lock level using **lock_before** D; declarations. C.**class**'s lock level is constructed to be above the lock levels of the classes mentioned in C's **lock_before** declarations. Cycles in the **lock_before** relation are not allowed. Also, the approach does not support specification of upper bounds for lock levels of *Class* objects.
- A class may declare some of its static fields as **rep**. The objects pointed to by a class's non-null *rep* fields are its *rep* objects. A class may declare a static invariant, which may depend on the class's static fields and the fields of its transitive *rep* objects. A *Class* object has an *inv* field, and the **pack** and **unpack** operations apply to *Class* objects as well as other objects. A *Class* object must be valid when it is shared and when it is unlocked.

We prevent deadlocks involving class initialization by applying the locking order to **initialize** operations as well, and by tracking static initializer executions in a thread's lock stack. Specifically, a thread is allowed to perform an **initialize** C; operation only if

- C.**class** is less than all objects on the thread's lock stack, or
- C.**class** is already shared (which implies the static initializer has already completed), or
- C.**class** is on the thread's lock stack (which implies that either the class is locked and therefore already shared or the thread is already executing the static initializer)

```
static_lockprotected class Counter {
  static int count;
  static invariant 0 ≤ count;

  static {
    // initialize Object;
    pack Counter.class;
    // share_lockprotected Counter.class;
  }

  static void increment()
    requires Counter.class < tid.lockStack;
  {
    // initialize Counter;
    synchronized (Counter.class) {
      unpack Counter.class;
      // initialize Counter;
      int c := count;
      // initialize Counter;
      count := c + 1;
      pack Counter.class;
    }
  }
}
```

Fig. 9. Example illustrating the programming model's support for static fields and static initializers. Commands implicitly inserted by the model are shown in comments.

It follows that on entry to the static initializer, we have

$$C.\mathbf{class} < \mathbf{tid}.\mathit{lockStack}$$

. Object C.**class** is pushed onto the lock stack for the duration of the static initializer's execution. Class C's static initializer may lock and then access classes that are less than C in the locking order.

An **initialize** C; operation ensures that if C.**class** is less than the objects on the lock stack, then in the post-state, C.**class** is shared.

An **initialize** operation's frame condition states that it does not modify any fields of any objects in the thread's access set.

Notice that in this approach, a thread must trigger an **initialize** C; operation before it can acquire the lock of class C. The easiest way to achieve this is by acquiring the lock inside a method of class C.

Figure 9 illustrates the approach. It shows how an **initial-ize** command is inserted at the top of each static method and before each static field access. Validity

```
static_immutable class Primes {
  static rep int[] primes;
  static invariant primes ≠ null ∧
     forall{int i in (0..primes.length − 2);
        primes[i] < primes[i + 1]};

  static {
     // initialize Object;
     primes := new int[] {2, 3, 5, 7, 11};
     pack Primes.class;
     // share_immutable Primes.class;
  }

  static int getThirdPrime()
     requires Primes.class < tid.lockStack;
  {
     // initialize Primes;
     read (Primes.class) {
        // initialize Primes;
        return primes[2];
     }
  }
}
```

Fig. 10. Example illustrating the support for classes with immutable static state.

of the **synchronized** command in method *increment* requires that *Counter*.**class** is shared. The preceding **initialize** command guarantees this, provided that *Counter*.**class** is not on the lock stack. This, in turn, is guaranteed by *increment*'s precondition.

6.3 **lock_before** *acyclicity*

The soundness of our approach requires that the **lock_be-fore** relation is acyclic. If a run-time system ensures that the module import relation is acyclic, then acyclicity of the **lock_before** relation may be ensured by checking at compile time that the **lock_before** relation on the classes of the module being compiled is acyclic.

However, neither the Java virtual machine nor the Microsoft .NET Framework's CLR refuse to load modules that import each other. As a result, the modules themselves are responsible for detecting cycles in the locking order at run time.

In our approach, this is achieved by requiring the program to build a module lock order graph, whose nodes are modules, at run time. The graph is stored in a static field in a class called *LockOrder* in a special module that all modules of the program must import. The graph is initially empty. Whenever a thread requires a lock order edge between a class C in a module M_1 and a class D in a different module M_2 , it must request it by performing a call

$$\mathit{LockOrder}.\mathit{checkEdge}(C.\mathbf{class}, D.\mathbf{class})$$

. This call first checks if a path already exists from M_1 to M_2. If so, the call returns normally. Otherwise, it checks if an edge from M_1 to M_2 would create a cycle. If so, the call throws an exception. Otherwise, the call adds the edge to the graph and returns normally.

6.4 Immutable Class Objects

The approach of the previous sections supports classes whose static fields are protected by locks. It is easy to extend the approach with more efficient support for classes whose static state is not modified after initialization, by allowing the immutable objects approach of Section 5 to be applied to *Class* objects as well.

A class must declare whether its *Class* object is shared as lock-protected or as immutable. Depending on this declaration, the implicit **share** operation at the end of the static initializer is either a **share_lockprotected** or a **share_immut-able** operation.

An **initialize** C; operation ensures that if C.**class** is less than the objects on the lock stack, then in the post-state, C.**class** is shared as declared.

Figure 10 illustrates the approach. Recall that validity of the array element access in method *getThirdPrime* requires that the array is in the read set. The array is inserted into the read set by the **read** command (see Section 5). The **read** command, in turn, requires that *Primes*.**class** is in the read set. This follows from the fact that it is immutable. The **initialize** command preceding the **read** command guarantees that *Primes*.**class** is immutable, provided that it is not on the lock stack. This, finally, is guaranteed by method *getThirdPrime*'s precondition.

7 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 postconditions, and assert statements declared by the program hold
- The program is free from null dereferences, array index out of bounds errors, and typecasting errors

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

- 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 [5].

The prototype verifier and the sample programs are available at http://www.cs.kuleuven.be/~bartj/.

8 Related Work

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

Method specifications in our methodology pertain only to the pre-state and post-state of method calls. Some systems [17,10] 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.* [5] 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 [7], by way of a type system [9], or using a theorem prover [18]. 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 [9] 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 [9] 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.* [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 [11].

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

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 [20].

Leino and Müller [16,15] propose an approach for verification of programs with static class invariants. Contrary to our work, they support neither multithreading nor Java and C#'s lazy class initialization semantics. Also, our approach is more flexible in terms of method effect framing w.r.t. static fields. Furthermore, our **lock_before** relation improves on the import order of [16], in that a) we do not restrict accessing classes as such, and b) contrary to the class import order, the **lock_before** order is consistent with the module import order, which is easier to understand for deployers and checking its acyclicity at run time is more efficient.

The present approach evolved from [12] and [14]. [14] improved upon [12] by supporting standard locking primitives, by preventing deadlocks, by supporting immutable objects, and by reporting on experience gained using a prototype verifier. This paper improves on [14] by adding support for static fields and static initializers.

9 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 [13] for an earlier version of the approach.

We are currently further extending the programming model to encompass lock re-entry and read-write locks.

References

[1] E. Ábrahám-Mumm, F. S. de Boer, W.-P. de Roever, and M. Steffen. Verification for Java's reentrant multithreading concept. In *FoSSaCS 2002*, volume 2303 of *LNCS*, pages 5–20. Springer, Apr. 2002.

[2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. 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] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

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

[5] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA 2002*, volume 37 of *SIGPLAN Notices*, pages 211–230. ACM, Nov. 2002.

[6] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, Dec. 1998.

[7] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *POPL 2004*, volume 39 of *SIGPLAN Notices*, pages 256–267. ACM, Jan. 2004.

[8] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI 2002*, volume 37 of *SIGPLAN Notices*, pages 234–245. ACM, May 2002.

[9] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI 2003*, pages 338–349. ACM, 2003.

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

[11] C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, Oct. 1974.

[12] B. Jacobs, K. R. M. Leino, F. Piessens, and W. Schulte. Safe concurrency for aggregate objects with invariants. In *Proc. Int. Conf. Software Engineering and Formal Methods (SEFM 2005)*, pages 137–146. IEEE Computer Society, sep 2005.

[13] B. Jacobs, K. R. M. Leino, F. Piessens, and W. Schulte. Safe concurrency for aggregate objects with invariants: Soundness proof. Technical Report MSR-TR-2005-85, Microsoft Research, jun 2005.

[14] B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A statically verifiable programming model for concurrent object-oriented programs. In *Proc. ICFEM*, 2006. To appear.

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

[16] K. R. M. Leino and P. Müller. Modular verification of static class invariants. In *Proc. Formal Methods (FM 2005)*, 2005.

[17] S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *POPL 2004*, volume 39 of *SIGPLAN Notices*, pages 245–255. ACM, Jan. 2004.

[18] E. Rodríguez, M. Dwyer, C. Flanagan, J. Hatcliff, G. T. Leavens, and Robby. Extending sequential specification techniques for modular specification and verification of multi-threaded programs. In *ECOOP 2005*, volume 3586 of *LNCS*, pages 551–576. Springer, July 2005.

[19] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.

[20] A. Welc, S. Jagannathan, and A. L. Hosking. Transactional monitors for concurrent objects. In *ECOOP 2004*, volume 3086 of *LNCS*. Springer, June 2004.