VeriCool: An Automatic Verifier for a Concurrent Object-Oriented Language

Jan Smans, Bart Jacobs, and Frank Piessens

Katholieke Universiteit Leuven, Belgium

Abstract. Reasoning about object-oriented programs is hard, due to aliasing, dynamic binding and the need for data abstraction and framing. Reasoning about *concurrent* object-oriented programs is even harder, since in general interference by other threads has to be taken into account at each program point.

In this paper, we propose an approach to the automatic verification of concurrent Java-like programs. The cornerstone of the approach is a programming model, a set of rules, which limits thread inference to synchronization points such that one can reason sequentially about most code. In particular, programs conforming to the programming model are guaranteed to be data race free. Compared to previous incarnations of the programming model, our approach is more flexible in describing the set of memory locations protected by an object's lock. In addition, we combine the model with an approach for data abstraction and framing based on dynamic frames. To the best of our knowledge, this is the first paper combining dynamic frames and concurrency.

We implemented the approach in a tool, called VeriCool, and used it to verify several small concurrent programs.

1 Introduction

In recent years, multi-processor and multi-core computers have become a commodity. To leverage the power provided by these multi-processor machines within a single application, developers must resort to multithreading. However, writing correct multithreaded programs is challenging. First of all, the non-determinism caused by thread scheduling makes finding errors through testing much less likely. Moreover, even when anomalies show up, they can be hard to reproduce. Secondly, reasoning about concurrent programs is hard, since in general interference by concurrently executing threads has to be taken into account at each program point. In particular, when threads concurrently access a shared data structure, special care has to be taken to avoid data races.

A data race occurs when two threads simultaneously access a shared memory location, and at least one of these accesses is a write access. Developers typically consider data races to be errors, since races can lead to hard-to-find bugs, and because they give rise to counter-intuitive, non-sequentially consistent executions under the Java memory model [1].

A simple strategy to prevent data races is to enclose each field access $o.f$ within a **synchronized**(o) block. Although this strategy is safe and rules out

G. Barthe and F. de Boer (Eds.): FMOODS 2008, LNCS 5051, pp. 220–239, 2008.

non-sequentially consistent executions, it is rarely used in practice, since it incurs a major performance penalty, is verbose, and only prevents low-level races. Instead, standard practice is to only lock objects that are effectively shared among multiple threads. However, it is difficult to determine which objects are meant to be shared and what locations are protected by an object's lock based solely on the program text. Therefore, it is hard for a compiler to determine whether the synchronization performed by the program suffices to rule out data races.

In this paper, we propose a programming model (a set of rules) such that programs that conform to the model contain no data races. In addition, we define a set of annotations to make the use of the programming model explicit. For example, a developer can annotate his code to make explicit whether an object is meant to be shared or not. Moreover, we explain how based on these annotations one can modularly and automatically verify whether a given program conforms to the model.

In summary, the contributions of this paper are as follows:

- We propose an approach to the automatic verification of concurrent Java-like programs. The cornerstone of the approach is a programming model for preventing data races. Compared to previous incarnations of the programming model [2,3,4], the approach is more flexible in describing the locations protected by an object's lock. The additional flexibility allows us to verify programs which cannot be verified in [2,3,4].
- To support data abstraction and framing, the approach relies on an existing solution based on dynamic frames [5,6]. A key insight of this paper is that dynamic frames can not only be used for abstract framing, but also to abstract over the locations protected by an object's lock. To the best of our knowledge, this is the first paper that combines concurrency and the dynamic frames approach.
- We implemented the approach in a tool, and used it to verify several concurrent programs. The verifier can be downloaded from the authors' homepage [7].

The remainder of this paper is structured as follows. In Section 2, we describe a programming model that ensures data race-freedom, and a way to verify whether a given Java program conforms to the model. In Section 3, we extend the approach of Section 2 with support for data abstraction. Finally, we compare with related work and conclude in Sections 4 and 5.

2 Preventing Data Races

In this section, we present the programming model that rules out data races (2.1), describe the annotations needed for modular verification (2.2), show how to statically verify whether an annotated program conforms to the model (2.3), and finally we explain why this verification approach works (2.4).

2.1 Programming Model

Our programming model prevents data races by ensuring that no two threads can ever access the same memory location concurrently. More specifically, we conceptually associate with each thread an access set, i.e. a set of memory locations that the thread can read and write, and the model guarantees that access sets of different threads are disjoint at all times. In our model, a location consists of an (object reference, field name) pair, and the location corresponding to $o.f$ is denoted $\&o.f$. The access set of a thread can grow and shrink over time. More specifically, four operations can affect a thread's access set:

- **Object creation.** When a thread t creates a new object o, all locations corresponding to fields of o are added to t's access set. For example, consider the constructor of the class *Counter* of Figure 1. At the beginning of this constructor, the field *count* of the newly created *Counter* object is made accessible to the current thread, and therefore it is safe to assign to the field within the body of the constructor.
- **Object sharing.** In addition to the accessibility of each location, the programming model also tracks each object's *sharedness*. That is, the model

```
final class Counter {
  int count;

  monitorfootprint
    { &count };
  monitorinvariant
    acc(count) ∧ 0 ≤ count;

  Counter()
    ensures acc(count);
    ensures ¬this.shared;
    ensures this.count = 0;
  {
    count = 0;
  }
}
```

```
class Session implements Runnable {
  shared Counter counter;

  Session(Counter c)
    requires c.shared;
    ensures acc(counter);
    ensures ¬this.shared;
  { counter = c; }

  void run()
    requires acc(this.∗);
    requires ∀{Object o • ¬o.locked};
  {
    synchronized(counter){
      counter.count + +;
    }
  }
}
```

```
Counter c = new Counter();
share c;
new Thread(new Session(c)).start();
new Thread(new Session(c)).start();
```

Fig. 1. A small concurrent program. The main program creates a new *Counter* object c, shares it, and then creates two threads. Each of these threads increments the counter within a **synchronized** block.

distinguishes *unshared* from *shared* objects. An unshared object is not meant to be locked, and a program violates the model if it attempts to do so. Conceptually, an unshared object has no corresponding lock. A shared object on the other hand can be locked by any thread.

The set of locations protected by a shared object's lock is called that object's *monitor footprint*. Our approach does not force the lock of an object o to protect all of o's fields, and therefore o's monitor footprint does not necessarily contain all fields of o. An object's *monitor invariant* is a predicate over its monitor footprint. Immediately after acquiring an object's lock, one may assume that the monitor invariant holds, and vice versa when releasing the lock one must establish the monitor invariant. For example, the monitor invariant of a *Counter* object o (Figure 1) states that the location corresponding to the field $o.count$ is accessible, and that the field holds a positive value.

Initially, new objects are unshared. When the execution of a thread t encounters a **share** o; statement[1], o transitions from the unshared to the shared state, provided o is not shared yet, o's monitor footprint is accessible to t, and o's monitor invariant holds. In addition, the sharing thread t loses access to all locations in o's monitor footprint. This is the only way an unshared object can become shared. After sharing, a thread must lock o to gain access to the locations in o's monitor footprint. Once an object is shared, it can never revert to the unshared state.

- **Acquiring and releasing locks.** After sharing an object o, threads can attempt to acquire o's lock. Whenever a thread t acquires this lock (e.g. in Java when t enters a **synchronized**(o) block), o's monitor footprint becomes accessible to t. Moreover, t may assume that o's monitor invariant holds over the locations in o's monitor footprint immediately after the acquisition of o's lock. When t decides to release o's lock, the thread again loses access to all locations in o's monitor footprint. Moreover, the programming model enforces that t can only release the lock when the monitor invariant holds.

  Note that an object's monitor footprint can grow and shrink over time. In particular, the monitor footprint at the time of locking does not have to equal the footprint at the time of release. This way accessibility of locations can be transferred from one thread to another.
- **Thread creation.** Starting a new thread transfers the accessibility of the locations corresponding to fields 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 .NET) from the starting thread to the started thread. For example, starting a new thread with the *Session* object of Figure 1 transfers accessibility of the object's *counter* field from the starting thread to the started thread.

The programming model described above is similar to the techniques used in various extensions of separation logic [8,9,10,11]. We discuss them in Section 4.

[1] The **share** statement is a special annotation which can appear in the body of a program. It is discussed in more detail in Section 2.2.

2.2 Annotations

Executions of programs that conform to the programming model contain no data races. However, without annotations it is difficult to verify whether a program conforms to the model. For example, in general it is undecidable to determine based on the program text whether an object is always shared or unshared at a given program point. In addition, we want to perform *modular* verification. That is, the correctness of a method implementation must not depend on implementation details of other methods and modules. Therefore, we propose a set of annotations that make the use of the programming model explicit and enable modular verification. We explain each annotation by means of the example of Figure 1.

Method Contracts. Each method has a corresponding method contract, consisting of a precondition and a postcondition. Both the precondition and the postcondition are boolean side-effect free expressions. The former define valid method pre-states, while the latter define valid method post-states. An expression is side-effect free if it contains no object or array creations, simple or compound assignments, and contains no method invocations[2]. Only parameters and the variable **this** may occur free in preconditions. Postconditions may additionally mention the variable **result**, denoting the return value of the method. Furthermore, postconditions may contain old expressions **old**(e), denoting the value of the expression e in the method's pre-state.

Class Contracts. Each class has a corresponding class contract, consisting of a monitor footprint and a monitor invariant. The monitor footprint of a class C is a side-effect free expression of type **set** which defines the set of locations protected by objects with dynamic type C. An expression of type **set** represent a set of memory locations. Each memory location is a (object reference, field name) pair, and the location corresponding to $o.f$ is denoted $\&o.f$. For example, the lock of a *Counter* object c protects the singleton $\{ \&c.count \}$. The monitor invariant of a class C is a boolean, side-effect free expression, and the programming model ensures that it can be assumed to hold on entry of a **synchronized**(o) block (where o has dynamic type C), provided the invariant is established again at the end of each synchronized block. For example, the monitor invariant of *Counter* states that the *count* field is accessible and that it holds a positive value. Only the variable **this** may occur free in class contracts. An omitted monitor footprint defaults to the empty set, while an omitted monitor invariant defaults to *true*.

Ghost State. The programming model tracks for each object whether it is shared or unshared, locked or unlocked, and for each location which thread is allowed to access the location. To express these properties, we introduce three special expressions. *o.shared* indicates whether o is shared. Similarly, *o.locked* indicates whether o is locked by the current thread. Hence, the second precondition of the

[2] In Section 3, we relax the definition of side-effect free expression by allowing invocations of pure methods.

method *run*, which is the first method executed by a new thread, states that the thread holds no locks. Finally, the boolean expression $\mathbf{acc}(o.f)$ denotes whether the location $\&o.f$ is accessible to the current thread. All three expressions can only be used within monitor invariants, preconditions and postconditions. In particular, they cannot appear within method implementations. This restriction ensures that annotations are erasable, i.e. that they can be omitted without affecting the execution of the annotated program. The expression $\mathbf{acc}(o.*)$ is syntactic sugar for stating that all locations corresponding to fields of o are accessible.

Share Statement. A developer can indicate that an unshared object o should transition to the shared state via the **share** o; statement. For example, the code snippet at the bottom of Figure 1 creates a new *Counter* object c and shares it. **share** is a ghost statement which only affects ghost state. As such, it can be erased without affecting the original program.

Shared Modifier. A field can be annotated with a **shared** modifier, indicating that it can only hold *null* or a reference to a shared object. The field *counter* of the class *Session* is an example of such a field. In the next section, we store information about the sharedness of fields in invariants, and we no longer need this modifier.

The program of Figure 1 is correctly synchronized, and the annotations enable the static verifier to prove this. In the next subsection, we explain how verification works.

2.3 Verification

Our verifier takes an annotated program as input and generates, via a translation into an intermediate verification language called BoogiePL [12], a set of verification conditions. The verification conditions are first-order logical formulas whose validity implies the correctness of the program. The formulas are analyzed automatically by satisfiability-modulo-theory (SMT) solvers. Our approach is based on a general approach described in [13]. In this subsection, we focus on novel aspects of our approach: namely the modeling of accessibility, lockedness and sharedness, the tracking of monitor footprints and monitor invariants, and the translation of statements and expressions to BoogiePL in a way that guarantees compliance with the programming model.

Notation. Heaps are modeled in the verification logic as maps from object references and field names to values. For example, the expression $h[o, f]$ denotes the value of the field f of object o in heap h. The function *wf* returns whether a given heap is well-formed, i.e. whether the fields of allocated objects point to allocated objects. H denotes the current value of the global heap. Allocatedness of objects is tracked by means of a special boolean field named *alloc*. The function *typeof* returns the dynamic type of a given object reference.

$\mathsf{Tr}^{E}_{h_1,h_2}[\![e]\!]$ denotes the translation of the side-effect free expression e to an equivalent first-order, BoogiePL expression, in a context where h_1 denotes the

current value of the heap and h_2 denotes the value of the old heap. Similarly, $\mathsf{Df}^E_{h_1,h_2}[\![e]\!]$ denotes the definedness of the side-effect free expression e, in a context where h_1 denotes the current value of the heap and h_2 denotes the value of the old heap. For example, the definedness of the expression x/y is given by $y \neq 0$. We will omit the value of the old heap when translating single-state expressions. Finally, $\mathsf{Tr}^S[\![s]\!]$ denotes the translation of the statement s to a number of BoogiePL statements.

Ghost State. The programming model tracks for each location whether it is accessible or not. In the verification logic, the accessibility of the location $\&o.f$ in a heap h is denoted $h[o, \mathit{accessible}][f]$. That is, $h[o, \mathit{accessible}]$ is a map from field names to booleans where each entry indicates whether the corresponding location is accessible. In addition, the programming model divides the set of object references into shared and unshared objects, and it further subdivides the set of shared objects into locked and unlocked objects. In the verification logic, an object is shared if $h[o, \mathit{shared}]$ is true. Similarly, an object is locked by the current thread if $h[o, \mathit{locked}]$ is true.

Monitor Footprints and Invariants. The programming model associates with each object a monitor footprint and a monitor invariant. To model this association, the verification logic contains two function symbols: *monitorfootprint* and *monitorinvariant*. For instance, $\mathit{monitorfootprint}(o)$ returns the set of locations protected by o's lock. To connect these function symbols to the class contracts, we introduce an axiom per class. More specifically, for each class C with monitor footprint F and monitor invariant I, the verification logic contains the following axiom:

$$\textbf{axiom}\ (\forall h, \mathit{this} \bullet \mathit{wf}(h) \wedge h[\mathit{this}, \mathit{alloc}] \wedge \mathit{this} \neq \mathit{null} \wedge \mathit{typeof}(\mathit{this}) = C \Rightarrow \mathit{monitorfootprint}(h, \mathit{this}) = \mathsf{Tr}^e_h[\![F]\!] \wedge \mathit{monitorinvariant}(h, \mathit{this}) = \mathsf{Tr}^e_h[\![I]\!]);$$

The class contract of a class C with monitor footprint F and monitor invariant I is well-formed only if the following conditions hold: (1) I is well-defined (i.e. $\forall h, \mathit{this} \bullet \mathit{wf}(h) \Rightarrow \mathsf{Df}^E_h[\![I]\!]$), (2) F is well-defined provided I holds, (3) F only contains accessible locations assuming I holds, (4) I only requires accessibility of locations in F assuming that I holds, and (5) both I and F are framed by F provided I holds, that is any heap modification outside of F does not affect the values of I and F.

Translation of Statements and Expressions. The programming model consists of a set of rules that together guarantee the absence of data races. To verify whether a program complies with the model, we have to update the ghost state tracked by the model after each statement, and check that statements do not violate the model. Figures 3, 5 and 6 of Appendices A and C contain the translation to BoogiePL for key statements and expressions. In this section, we shortly highlight two important aspects of the translation.

A central rule in the programming model is that threads can only access locations in their corresponding access set. To enforce this restriction, each field access (both read and write) is preceded by a proof obligation requiring $\&o.f$ to be accessible the current thread.

In general, interference by concurrently executing threads has to be taken into account at each program point. However, our programming model enforces that threads can only read and write accessible locations. Moreover, the accessibility of a location with respect to a certain thread t can only be changed by t itself. Therefore, we only need to take into account the effect of other threads at synchronization points. More specifically, in the translation to BoogiePL, the effect of other threads is modeled by havocing (i.e. assigning non-deterministically chosen values to) locations that are added or removed from a thread's access set at share and lock statements.

Method Contract Validity. Programming errors can not only show up in method implementations, but also in method contracts. To ensure that contracts are meaningful, we check that they are well-defined. For a method m with precondition P, postcondition Q and parameters $x_1, \ldots x_n$ in a class C, the following proof obligation is generated:

$$\textbf{assert } (\forall h_{old}, h, \mathit{this}, x_1, \ldots x_n \bullet \mathit{wf}(h_{old}) \wedge \mathit{wf}(h) \wedge h[\mathit{this}, \mathit{alloc}] \wedge h_{old}[\mathit{this}, \mathit{alloc}] \wedge \mathit{this} \neq \mathit{null} \wedge \mathit{typeof}(\mathit{this}) <: C \Rightarrow \mathsf{Df}^E_{h_{old}}[\![P]\!] \wedge (\mathsf{Tr}^E_{h_{old}}[\![P]\!] \Rightarrow \mathsf{Df}^E_{h,h_{old}}[\![Q]\!]));$$

2.4 Soundness

In order to prevent data races, (1) threads should only be able to access locations in their access set, and (2) thread access sets and the monitor footprints of shared, non-locked objects must partition the set of allocated locations.

Since each field access is guarded by an accessibility check (see Figures 3 and 5), property (1) follows immediately. We outline a proof by induction on the length of the execution trace for property (2). In the initial state, the set of allocated locations is empty, the main thread's access set is empty, and no objects are shared yet. Therefore, property (2) holds in the initial state. Now assume that property (2) holds in a state σ. We have to demonstrate that each statement s leading from σ to a state σ' preserves (2). We consider two cases. The other cases are similar.

- Assume that s is a **share** o; statement. Successful verification ensures that o is non-null, unshared and that its monitor invariant holds in state σ. The well-formedness of the class contract implies that o's monitor footprint contains only locations that are accessible to the current thread. Immediately after the share statement, o is shared and the current thread can no longer access the locations in o monitor footprint. Property (2) is preserved, since the access sets of other threads are not affected, the footprints of shared, non-locked objects are not affected, and the old access set of the current thread is split in the new access set and the locations in o's monitor footprint.
- Assume that s is a lock acquisition (**synchronized**(o)). The program verified successfully, and hence we may assume that o was shared but not locked by the current thread before entering the synchronized block. Moreover, the Java semantics guarantee that upon entering the synchronized block no other thread was holding o's lock. As a consequence, we may assume that

o's monitor footprint is disjoint from any thread's access set in state σ. By adding the monitor footprint to the current thread's access set, and by making o locked, we preserve property (2) in σ'.

By induction, we may conclude that programs that verify only give rise to states where (2) holds.

3 Data Abstraction

Data abstraction is crucial in the construction of modular programs, since it ensures that internal changes in one module do not propagate to other modules. In object-oriented programs, classes typically enforce data abstraction by providing access to their internal state only through methods.

The class *Counter* in Figure 1 however was not written with data abstraction in mind, since it directly exposes its internal *count* field to client code. As a consequence, any change in *Counter*'s implementation forces us to rewrite or at least reconsider the correctness of client code. For example, renaming the *count* field breaks the implementation of *Session*'s *run* method.

Recently, we proposed an approach to the automatic verification of sequential, object-oriented programs that use method calls in specifications for data abstraction [5]. In this approach, methods used for this purpose have to be side-effect free, and are called pure methods. To solve the framing problem, that is to determine the effect of field assignments and executions of non-pure methods on the return values of pure methods, the approach relies on method footprint annotations, which specify an upper bound on the memory locations read (in case of pure methods) or written (in case of non-pure methods) by the corresponding method. More specifically, to prove that a state change (i.e. field update or non-pure method invocation) does not affect the return value of a pure method, one has to show that the footprint of the state change is disjoint from the pure method's footprint. Thanks to the use of dynamic frames [6], special pure methods that return a set of memory locations, method footprints can be specified without breaking data abstraction.

As an example, consider the class *Counter* of Figure 2. Contrary to the older version of the class of Figure 1, client code can only access the internal field *count* through the setter *increment* and the getter *getCount*. Furthermore, *Counter*'s method contracts do not mention the field *count*, and instead rely on public, pure methods to specify the behavior. In particular, both pure and non-pure methods define the locations they read or write in terms of the pure method *rep*, a dynamic frame. Since client code only depends on *Counter*'s public interface, changing the class's internal representation does not affect them. For example, renaming the field *count* would not endanger the correctness of the class *Session*.

In the approach described in the previous section, the monitor footprint of a class C specifies the set of locations protected by locks of objects with dynamic type C. For example, the class contract of *Counter* (Figure 1) specifies that the

lock of a *Counter* object o protects the singleton { $\&o.count$ }. However, this class contract exposes the internal field *count*. A key insight of this paper is that dynamic frames can not only be used to abstractly specify the locations read or written by a method, but also to abstract over the locations protected by an object's lock. Indeed, the dynamic frame *rep* can be used to specify *Counter*'s monitor footprint without revealing any internal fields, as shown in Figure 2. Similarly, the pure method *inv* can be used to abstractly specify the monitor invariant.

In summary, by combining the approach for verifying concurrent programs of the previous section with the solution for data abstraction and framing of [5], we can construct a verifier for concurrent programs that supports both data abstraction and framing. In the remainder of this section, we describe the extra annotations needed for dynamic frames, we highlight the most important changes in verification with respect to [5], and finally we sketch how our approach can be extended to deal with read-write locks.

3.1 Annotations

We extend the set of annotations with pure and predicate method modifiers and with method footprints.

Pure Methods and Predicates. A method can be annotated with a **pure** modifier, indicating that it can be used in specifications. The body of a pure method must consist of a single return statement returning a side-effect free expression. From now on, side-effect free expressions may contain method invocations but only to pure methods. Non-pure methods are called mutators. Pure methods with return type **set** are called dynamic frames. Purity is inherited by overriding methods.

Predicates are special pure methods marked with **predicate**. More specifically, a predicate is a boolean, pure method that can only be called from other predicates and within class and method contracts. Contrary to other method implementations, predicates are allowed to mention $\mathbf{acc}(o.f)$, $o.shared$, and $o.locked$.

Method Footprints. In addition to pre and postconditions, each method contract contains a method footprint. A method footprint is a side-effect free expression of type **set**. The footprint of a pure method (**reads** annotation) specifies the locations that can potentially be be read by the method, while a mutator's footprint (**writes** annotation) specifies the locations that can be modified by the method. More specifically, a mutator can only modify $o.f$ if $\&o.f$ is in the method's footprint or if o was unallocated at the start of the method. Only parameters and the variable **this** may occur free in method footprints.

To prove that a state change (i.e. field update or non-pure method invocation) does not affect the return value of a pure method, one has to show that the footprint of the state change is disjoint from the pure method's footprint. To preserve disjointness of footprints, constructors and mutators should specify how

```
final class Counter {
  private int count;

  monitorfootprint rep();
  monitorinvariant inv();

  Counter()
    writes ∅;
    ensures inv();
    ensures getCount() = 0;
    ensures elemsFresh(rep());
    ensures ¬this.shared;
  { }

  void increment()
    requires inv();
    writes rep();
    ensures inv();
    ensures getCount() = old(getCount()) + 1;
    ensures newElemsFresh(rep());
  { count++; }

  pure int getCount()
    requires inv();
    reads rep();
  { return count; }

  predicate bool inv()
    reads inv()?rep() : universe;
  { return acc(count) ∧ 0 ≤ count; }

  pure set rep()
    requires inv();
    reads rep();
  { return { &count }; }
}
```

```
class Session implements Runnable {
  private Counter counter;

  Session(Counter c)
    requires c.shared;
    writes ∅;
    ensures inv();
    ensures elemsFresh(rep());
    ensures ¬this.shared;
  { counter = c; }

  void run()
  {
    synchronized(counter){
      counter.increment();
    }
  }

  predicate bool inv()
  { return acc(counter) ∧ counter ≠ null∧
      counter.shared; }

  pure set rep()
  { return { &counter }; }
}
```

```
Counter c = new Counter();
share c;
new Thread(new Session(c)).start();
new Thread(new Session(c)).start();
```

Fig. 2. A revision of the program of Figure 1. The classes *Counter* and *Session* now both hide their internal fields, and instead provide methods to query and update their state.

they affect footprints. To this end, we introduce two new boolean expressions: **elemsFresh**(e) and **newElemsFresh**(e). Both expressions can only be used in postconditions. **elemsFresh**(e) states that the locations in e are fresh with respect to the method pre-state, while **newElemsFresh**(e) states that each location in e is either an element of **old**(e) or is fresh with respect to the method pre-state.

The program of Figure 2 successfully verifies. The contracts for the classes *Runnable* and *Thread* which are needed for verification are shown in Figure 4 of appendix B. Note that the methods *run*, *inv* and *rep* in class *Session* override the corresponding methods in *Runnable*, and therefore they inherit the contracts of their overridden methods.

3.2 Verification

We shortly highlight the most important changes in the verification approach with respect to [5]. For details on the encoding of pure methods and method footprints, we refer the reader to [5].

In [5], a (postcondition) axiom is generated for each dynamic frame, stating that the set returned by the method only contains allocated locations. In this paper, we generate an additional axiom stating that dynamic frames only return sets containing accessible locations. This axiom is used to deduce that the monitor footprint of a newly acquired object does not overlap with any locations accessible to the thread before entering the synchronized block.

Furthermore, whenever a predicate method requires an object to be accessible, that is, whenever it contains a subexpression $\mathbf{acc}(o.f)$, we require $\&o.f$ to be an element of the method's reads clause. This allows predicates to be used in monitor invariants.

3.3 Read-Write Locks

A read-write lock is a variant of a traditional lock, which may be concurrently held by multiple reader threads, as long as there are no writers. For writer threads, the read-write lock is exclusive. For example, in the program of appendix D a shared arraylist object is protected by a read-write lock. Multiple threads can concurrently iterate over the elements in the list by acquiring the read lock ($\mathbf{synchronized}_R$). In previous incarnations of the programming model, it is impossible to verify this program.

Supporting read-write locks requires only minor extensions to the programming model and the annotations. More specifically, instead of associating an access set with each thread, we associate with each thread a read and a write set, the set of locations that can respectively be read or written by the corresponding thread. $\mathbf{acc}(o.f)$ now means that $\&o.f$ is in both sets, while $\mathbf{read}(o.f)$ signifies that $\&o.f$ is at least in the current thread's read set. Moreover, each class contract is extended with a read monitor footprint and a read monitor invariant. The regular monitor invariant must imply the read monitor invariant, and similarly the regular monitor footprint must be a superset of the read monitor footprint. When acquiring the read lock of an object o, the locations in o's read monitor footprint are added to the thread's read set, and one may assume o's read monitor invariant holds. And vice versa, when releasing a read lock, one must establish the read monitor invariant, and the thread loses read permission for the locations in the read monitor footprint.

4 Related Work

The Extended Static Checker for Java [14] (ESC/Java) is a compile-time program checker that attempts to find common errors, such as null dereferences and data races, in Java programs. Similarly to VeriCool, ESC/Java relies on verification condition generation and theorem proving. However, the tool trades soundness for ease of use. For example, it assumes that the value of a shared location stays unchanged if a method releases and then reacquires the lock that protects it, ignoring the possibility that some other thread might have acquired the lock and modified the location in the interim [15, page 91].

In his thesis, Kassios [6] describes a flexible approach to data abstraction and framing based on dynamic frames. More specifically, Kassios uses specification variables, similar to our pure methods, to achieve data abstraction. To solve the framing problem, he proposes using dynamic frames to abstractly specify the footprint of specification variables and the effect of mutator methods. Recently, we showed how Kassios' ideas can be incorporated in a program verifier for a Java-like language based on first-order logic [5]. However, both [6] and [5] consider only sequential object-oriented programs. In this paper, we extend the solution described in [5] in order to handle concurrent programs. To the best of our knowledge, this is the first paper combining dynamic frames and concurrency.

Various extensions of separation logic to concurrent programs have been proposed [9,10,16,8,11]. In particular, Gotsman *et al.* [10] propose a variant of concurrent separation logic that supports an unbounded number of dynamically allocated locks and threads, which is similar to our approach in many respects. For example, the boolean expression $\mathbf{acc}(o.f)$ can be considered to be the counterpart of the separation logic predicate $o.f \mapsto \_$ in the sense that they both represent a permission to access the location $\&o.f$. One difference between their approach and ours is that they allow lock finalization, while we never allow a shared object to become unshared. To achieve the same effect in our approach though, one could temporarily wrap the unshared object in a shared object. Another difference is that we make footprints explicit (typically via dynamic frames), while footprints are implicit in separation logic. Gotsman *et al.* developed a detailed formalization and soundness proof, but their approach has not been implemented in an automatic verifier.

Smallfoot [16,17] is an automatic verifier for concurrent separation logic geared toward verification of concurrent programs that manipulate recursive data structures. Smallfoot can verify many highly concurrent programs that our tool cannot. However, the tool relies on various built-in rules about list and trees. Our tool has no built-in rules to reason about particular data structures, but instead each class can define its own abstractions via pure methods.

Ábrahám-Mumm *et al.* [18] propose an assertional proof method for Java's reentrant monitors. Their approach supports class invariants, but these invariants can only mention fields of this. Our approach has no such restriction.

A number of type systems have been proposed to prevent data races in object-oriented programs. For example, Boyapati *et al.* [19] parametrize 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, read-only object

and unique pointers. Our approach not only rules out data races, but also supports reasoning about richer properties such as object invariants.

The rules in our programming model that a lock's monitor invariant must hold whenever it is released and that it can be assumed to hold whenever it is acquired, are taken from Hoare's work on monitors [20].

This paper improves upon previous incarnations of the programming model [2,3,4]. First of all, the present approach is more flexible in specifying the locations protected by an object's lock. More specifically, [2,3,4] determine the contents of the monitor footprint by means of rep annotations on fields, i.e. all fields of all objects transitively reachable from an object o through rep fields are protected by o's lock. However, rep fields rule out sharing of memory locations (among different footprints). As a consequence, the concurrent iterator program of Figure 7 of appendix D, where different iterators share the locations in an array list's footprint, cannot be verified by [2,3,4]. Secondly, the present approach is more fine-grained in the sense that accessibility is tracked per location instead of per object. This implies that different fields of an object can be protected by different locks, which is not possible in previous versions. Finally, we demonstrate how to combine the programming model with an approach to data abstraction and framing based on dynamic frames. Data abstraction was not considered in [2,3,4].

5 Conclusion

We propose an approach to the automatic verification of concurrent Java-like programs. The cornerstone of the approach is a programming model for preventing both low-level and high-level races. Compared to previous incarnations of the model, we are more flexible in specifying the locations protected by an object's lock. In addition, we combine the model with an approach for data abstraction and framing based on dynamic frames [6,5]. To the best of our knowledge, this is the first paper that combines dynamic frames and concurrency. We implemented our approach in a tool, and used it to automatically verify several small concurrent programs.

In the future, we plan to apply our approach in a larger case study.

Acknowledgments

Jan Smans is a research assistant of the Fund for Scientific Research - Flanders (FWO). Bart Jacobs is a postdoctoral fellow of the Fund for Scientific Research - Flanders (FWO).

References

1. Gosling, J., Joy, B., Steele, G., Bracha, G.: The java language specification, 3rd edn. (2005)
2. Jacobs, B., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. In: SAVCBS (2004)

3. Jacobs, B., Leino, K.R.M., Piessens, F., Schulte, W.: Safe concurrency for aggregate objects with invariants. In: SEFM (2005)
4. Jacobs, B., Smans, J., Piessens, F., Schulte, W.: A statically verifiable programming model for concurrent object-oriented programs. In: ICFEM (2006)
5. Smans, J., Jacobs, B., Piessens, F., Schulte, W.: An automatic verifier for java-like programs based on dynamic frames (2008)
6. Kassios, Y.: A Theory of Object Oriented Refinement. PhD thesis, University of Toronto (2006)
7. `http://www.cs.kuleuven.be/~jans/vericool`
8. Hobor, A., Appel, A.W., Nardelli, F.Z.: Oracle semantics for concurrent separation logic. In: ESOP (2008)
9. O'Hearn, P.W.: Resources, concurrency and local reasoning. Theoretical Computer Science 375(1-3) (2007)
10. Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., Sagiv, M.: Local reasoning for storable locks and threads. In: APLAS (2007)
11. Haack, C., Hurlin, C.: Separation logic contracts for a java-like language with fork/join. Technical Report 6430, INRIA (2008)
12. DeLine, R., Leino, K.R.M.: Boogiepl: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-, -70 (2005)
13. Leino, K.R.M., Schulte, W.: A verifying compiler for a multi-threaded object-oriented language. In: Marktoberdorf Summer School Lecture Notes (2006)
14. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: PLDI (2002)
15. Leino, K.R.M., Nelson, G., Saxe, J.B.: Esc/java user's manual. Technical Report SRC-TN-2000-002, Compaq Research Center (2000)
16. Calcagno, C., Parkinson, M., Vafeiadis, V.: Modular safety checking for fine-grained concurrency. In: SAS (2007)
17. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: FMCO (2005)
18. Ábrahám Mumm, E., de Boer, F.S., de Roever, W.P., Steffen, M.: Verification for java's reentrant multithreading concept. In: Nielsen, M., Engberg, U. (eds.) ETAPS 2002 and FOSSACS 2002. LNCS, vol. 2303, Springer, Heidelberg (2002)
19. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: Preventing data races and deadlocks. In: OOPSLA (2002)
20. Hoare, C.: Monitors: An operating system structuring concept. cacm 17(10) (1974)

Appendix

A Translation of Expressions

$$
\begin{array}{l}
\mathsf{Tr}_h^E[\![e.f]\!] \equiv h[\mathsf{Tr}_h^E[\![e]\!], f] \\
\mathsf{Tr}_h^E[\![\mathbf{acc}(e.f)]\!] \equiv h[\mathsf{Tr}_h^E[\![e]\!], \mathit{accessible}][f] \\
\mathsf{Tr}_h^E[\![\{\ \&e.f\ \}]\!] \equiv \{\ (\mathsf{Tr}_h^E[\![e]\!], f)\ \} \\
\mathsf{Tr}_h^E[\![e.\mathit{shared}]\!] \equiv h[\mathsf{Tr}_h^E[\![e]\!], \mathit{shared}] \\
\mathsf{Tr}_h^E[\![e.\mathit{locked}]\!] \equiv h[\mathsf{Tr}_h^E[\![e]\!], \mathit{locked}] \\
\mathsf{Tr}_h^E[\![e.m(e_1,\ldots,e_n)]\!] \equiv \#C.m(h, \mathsf{Tr}_h^E[\![e]\!], \mathsf{Tr}_h^E[\![e_1]\!], \ldots, \mathsf{Tr}_h^E[\![e_n]\!])
\end{array}
$$

$$
\begin{array}{l}
\mathsf{Df}_h^E[\![e.f]\!] \equiv \mathsf{Df}_h^E[\![e]\!] \wedge \mathsf{Tr}_h^E[\![e]\!] \neq \mathit{null} \wedge h[\mathsf{Tr}_h^E[\![e]\!], \mathit{accessible}][f] \\
\mathsf{Df}_h^E[\![\mathbf{acc}(e.f)]\!] \equiv \mathsf{Df}_h^E[\![e]\!] \wedge \mathsf{Tr}_h^E[\![e]\!] \neq \mathit{null} \\
\mathsf{Df}_h^E[\![\{\ \&e.f\ \}]\!] \equiv \mathsf{Df}_h^E[\![e]\!] \wedge \mathsf{Tr}_h^E[\![e]\!] \neq \mathit{null} \wedge h[\mathsf{Tr}_h^E[\![e]\!], \mathit{accessible}][f] \\
\mathsf{Df}_h^E[\![e.\mathit{shared}]\!] \equiv \mathsf{Df}_h^E[\![e]\!] \wedge \mathsf{Tr}_h^E[\![e]\!] \neq \mathit{null} \\
\mathsf{Df}_h^E[\![e.\mathit{locked}]\!] \equiv \mathsf{Df}_h^E[\![e]\!] \wedge \mathsf{Tr}_h^E[\![e]\!] \neq \mathit{null} \\
\mathsf{Df}_h^E[\![e.m(e_1,\ldots,e_n)]\!] \equiv \mathsf{Df}_h^E[\![e]\!] \wedge \mathsf{Df}_h^E[\![e_1]\!] \wedge \ldots \mathsf{Df}_h^E[\![e_n]\!] \wedge \\
\qquad \mathsf{Tr}_h^E[\![e]\!] \neq \mathit{null} \wedge \mathsf{Tr}_h^E[\![\mathit{Pre}_m\,[e/\mathbf{this}, e_1/x_1, \ldots, e_n/x_n]]\!]
\end{array}
$$

Fig. 3. Translation and definedness of expressions

B Contracts for Runnable and Thread

```
class Thread {
  Thread(Runnable target)
    requires target ≠ null ∧ target.inv();
    writes target.rep();
    ensures inv();
    ensures elemsFresh(rep());
    ensures ¬this.shared;
  void start()
    requires inv();
    writes rep();
  predicate bool inv()
    reads inv()?rep() : universe;
  pure set rep()
    requires inv();
    reads rep();
}
```

```
interface Runnable {
  void run()
    requires inv();
    requires ∀{Object o • ¬o.locked};
  predicate bool inv()
    reads inv()?rep() : universe;
  pure set rep()
    requires inv();
    reads rep();
}
```

Fig. 4. The contracts for the interface *Runnable* and the class *Thread*

C Translation of Statements

We assume expressions nested within statements are side-effect free.

$\mathsf{Tr}^S[\![x = \textbf{new } C(e_1, \ldots, e_n);]\!] \equiv$
  assert $\mathsf{Df}^E_h[\![e_1]\!] \wedge \ldots \mathsf{Df}^E_h[\![e_n]\!]$;
  havoc *newObject*;
  assume $newObject \neq null \wedge \neg H[newObject, alloc] \wedge typeof(newObject) = C$;
  assume $(\forall q, f \bullet (q, f) \in locationsOf(newObject) \Rightarrow \neg H[q, accessible][f])$;
  $oldH := H$; **havoc** H; **assume** $successor(oldH, H)$;
  assume $(\forall q, f \bullet oldH[q, accessible][f] \vee (q, f) \in locationsOf(newObject) \Leftrightarrow$
    $H[q, accessible][f])$;
  assume $(\forall q, f \bullet oldH[q, accessible][f] \Rightarrow oldH[q, f] = H[q, f])$;
  assume $(\forall q \bullet H[q, locked] \Leftrightarrow oldH[q, locked])$;
  assume $H[newObject, alloc]$;
  call $C.ctr(newObject, \mathsf{Tr}^E_h[\![e_1]\!], \ldots, \mathsf{Tr}^E_h[\![e_n]\!])$;
  $x := newObject$;

$\mathsf{Tr}^S[\![e_1.f = e_2;]\!] \equiv$
  assert $\mathsf{Df}^E_H[\![e_1]\!] \wedge \mathsf{Df}^E_H[\![e_2]\!]$;
  assert $\mathsf{Tr}^E_H[\![e_1]\!] \neq null$;
  assert $H[\mathsf{Tr}^E_H[\![e_1]\!], accessible][f]$;
  $H[\mathsf{Tr}^E_H[\![e_1]\!], f] := \mathsf{Tr}^E_H[\![e_2]\!]$;

$\mathsf{Tr}^S[\![\textbf{share } e;]\!] \equiv$
  assert $\mathsf{Df}^E_H[\![e]\!]$;
  $sharedObject := \mathsf{Tr}^E_H[\![e]\!]$;
  assert $sharedObject \neq null \wedge \neg H[sharedObject, shared] \wedge monitorinvariant(H, sharedObject)$;
  $oldH := H$; **havoc** H; **assume** $successor(oldH, H)$;
  assume $(\forall q, f \bullet oldH[q, accessible][f] \wedge \neg(q, f) \in monitorfootprint(oldH, sharedObject) \Leftrightarrow$
    $H[q, accessible][f])$;
  assume $(\forall q, f \bullet oldH[q, accessible][f] \wedge \neg(q, f) \in monitorfootprint(oldH, sharedObject) \Rightarrow$
    $oldH[q, f] = H[q, f])$;
  assume $(\forall q \bullet H[q, locked] \Leftrightarrow oldH[q, locked])$;
  assume $H[sharedObject, shared]$;

Fig. 5. Translation of statements

$\mathsf{Tr}[\![\textbf{synchronized}(e)\{\ s\ \}]\!] \equiv$
  assert $\mathsf{Df}^E_H[\![e]\!]$;
  $lockedObject := \mathsf{Tr}^E_H[\![e]\!]$;
  assert $H[lockedObject, shared] \wedge \neg H[lockedObject, locked]$;
  $oldH := H$; **havoc** H; **assume** $successor(oldH, H)$;
  assume $monitorinvariant(H, lockedObject)$;
  assume $(\forall q, f \bullet oldH[q, accessible][f] \vee (q, f) \in monitorfootprint(H, lockedObject) \Leftrightarrow H[q, accessible][f])$;
  assume $(\forall q, f \bullet oldH[q, accessible][f] \Rightarrow oldH[q, f] = H[q, f])$;
  assume $(\forall q, f \bullet (q, f) \in monitorfootprint(H, lockedObject) \Rightarrow \neg oldH[q, accessible][f])$;
  assume $(\forall q \bullet q \neq lockedObject \Rightarrow H[q, locked] \Leftrightarrow oldH[q, locked])$;
  assume $H[lockedObject, locked]$;
  $\mathsf{Tr}^s[\![s]\!]$
  assert $monitorinvariant(H, lockedObject)$;
  $oldH := H$;
  havoc H;
  assume $successor(oldH, H)$;
  assume $(\forall q, f \bullet oldH[q, accessible][f] \wedge \neg(q, f) \in monitorfootprint(oldH, lockedObject) \Leftrightarrow H[q, accessible][f])$;
  assume $(\forall q, f \bullet oldH[q, accessible][f] \wedge \neg(q, f) \in monitorfootprint(oldH, lockedObject) \Rightarrow oldH[q, f] = H[q, f])$;
  assume $(\forall q \bullet q \neq lockedObject \Rightarrow H[q, locked] \Leftrightarrow oldH[q, locked])$;
  assume $\neg H[lockedObject, locked]$;

Fig. 6. Translation of statements (cont.)

D Concurrent Iterator

```
final class ArrayList {
  readmonitorfootprint rep_R();
  readmonitorinvariant inv_R();

  int count; Object[] items;
  ...
  pure Object get(int i);
    requires inv_R() ∧ 0 ≤ i < size();
    reads rep_R();
  { return items[i]; }

  pure int size();
    requires inv_R();
    reads rep_R();
    ensures 0 ≤ result;
  { return count; }

  predicate bool inv_R()
    reads inv_R()?rep_R() : universe;
  { return read(count) ∧ read(items)∧
    items ≠ null ∧ read(items.elems)∧
    0 ≤ count ≤ items.length; }

  pure set rep_R()
    requires inv_R();
    reads rep_R();
  { return {&count, &items} ∪ elems(items); }
}
```

```
class Iterator {
  ArrayList list; int index;
  ...
  Object next()
    requires inv() ∧ hasNext();
    writes rep();
    ensures inv() ∧ list() = old(list());
    ensures newElemsFresh(rep());
  { return list.items[index++]; }

  pure bool hasNext()
    requires inv();
    reads rep() ∪ list().rep_R();
  { return index < list.count; }

  pure ArrayList list()
    requires inv();
    reads rep();
  { return list; }

  predicate bool inv()
    reads inv()?
      (rep() ∪ list().rep_R()) : universe;
  { return list ≠ null ∧ list.inv_R()∧
    0 ≤ index ≤ list.count∧
    &list ∉ list.rep_R()∧
    &index ∉ list.rep_R(); }

  pure set rep()
    requires inv();
    reads rep();
  { return {&list, &index}; } }
```

Fig. 7. The iterator pattern. By using read locks, multiple threads can simultaneously iterate over a shared array list.

```
class Session implements Runnable {
  private ArrayList list;

  Session(ArrayList l)
    requires l.shared;
    writes ∅;
    ensures inv();
    ensures elemsFresh(rep());
    ensures ¬this.shared;
  { list = l; }

  void run()
  {
    synchronized_R(list){
      Iterator iter = new Iterator(list);
      while(iter.hasNext())
        loopinvariant iter.inv() ∧ newElemsFresh(iter.rep())∧
          iter.list() = old(iter.list());
        writes iter.rep();
      { iter.next(); }
    }
  }

  predicate bool inv()
  { return acc(counter) ∧ counter ≠ null ∧ counter.shared; }

  pure set rep()
  { return { &counter }; }
}
```

Fig. 8. The iterator pattern (cont)