A basis for verifying multi-threaded programs

K. Rustan M. Leino[0] and Peter Müller[1]

[0] Microsoft Research, Redmond, WA, USA, leino@microsoft.com
[1] ETH Zurich, Switzerland, peter.mueller@inf.ethz.ch

Abstract. Advanced multi-threaded programs apply concurrency concepts in sophisticated ways. For instance, they use fine-grained locking to increase parallelism and change locking orders dynamically when data structures are being reorganized. This paper presents a sound and modular verification methodology that can handle advanced concurrency patterns in multi-threaded, object-based programs. The methodology is based on implicit dynamic frames and uses fractional permissions to support fine-grained locking. It supports concepts such as multi-object monitor invariants, thread-local and shared objects, thread pre- and postconditions, and deadlock prevention with a dynamically changeable locking order. The paper prescribes the generation of verification conditions in first-order logic, well-suited for scrutiny by off-the-shelf SMT solvers. A verifier for the methodology has been implemented for an experimental language, and has been used to verify several challenging examples including hand-over-hand locking for linked lists and a lock re-ordering algorithm.

0 Introduction

Mainstream concurrent programs use multiple threads and synchronization through locks or monitors. To increase parallelism and to reduce the locking overhead, they apply these concurrency concepts in sophisticated ways. They use fine-grained locking to permit several threads to access a data structure concurrently. They distinguish between thread-local and shared objects to avoid unnecessary locking of thread-local objects, and they allow objects to transition from thread-local to shared and back. They dynamically change locking orders, which are used to prevent deadlocks, when data structures are being reorganized. They distinguish between read and write accesses to permit concurrent reading but ensure exclusive writing. Several other such concurrency patterns are described in the literature [16, 9].

These patterns improve the performance and flexibility of programs, but also complicate reasoning. For instance, fine-grained locking often requires that several locks be acquired before a field can be updated safely. Omitting one of the locks potentially leads to inconsistent data structures. Consider for example a sorted linked list, where each node has to maintain a monitor invariant such as $next \neq \textbf{null} \Rightarrow val \leq next.val$. Updating a field $n.val$ potentially breaks the invariant of n and n's predecessor in the list. Consequently, the monitors of both objects have to be acquired before updating $n.val$, and the monitor invariants of both monitors have to be checked when they are released. This problem does not occur with coarse-grained locking, where invariants over several objects can be associated with the (single) lock for the whole data structure. Other advanced concurrency patterns also lead to subtle correctness conditions, which is one of the reasons why concurrent programs are so difficult to get right.

A standard verification technique for concurrent programs is to proceed in two steps. First, the code is divided into atomic sections. Second, sequential reasoning is used within each atomic section and rely-guarantee reasoning [14, 20] between atomic sections. Advanced concurrency patterns complicate especially the first step because atomicity is not always achieved by acquiring a single lock; instead, the permission to access a field may be justified by thread-locality, by acquiring one or more locks, or by sharing fields just among readers.

In this paper, we present a verification methodology for multi-threaded, object-based programs that handles all of these complications. It verifies the absence of data races and deadlocks, and that implementations satisfy their contracts. We build on Smans *et al.*'s implicit dynamic frames [18] and extend them to concurrent programs. Contracts such as monitor invariants specify access permissions along with conditions on variables. Evaluating these contracts transfers these access permissions, for instance from a monitor to the thread that acquires the monitor. To support fine-grained locking and concurrent reading, we use Boyland's fractional permissions [4], which allow us to split the access permission for a field among several monitors or threads. The permission accounting is similar to previous work on concurrent separation logic [2, 8], but our approach generates verification conditions in first-order logic, well-suited for off-the-shelf SMT solvers such as Z3 [6]. Finally, our methodology permits not a fixed but a changeable locking order among monitors. We have implemented a verifier for our methodology for an experimental language called Chalice, and have used it to verify automatically several challenging examples including hand-over-hand locking for linked lists and a lock re-ordering algorithm.

Outline. The next three sections present our verification methodology informally: Section 1 explains permissions, Section 2 discusses shared objects and thread synchronization, and Section 3 shows how we prevent deadlocks. The formal encoding including proof obligations is presented in Section 4. We discuss related work in Section 5 and conclude in Section 6.

1 Permissions

A thread may access a heap location only if it has the permission to do so. Abstractly, a *permission* is a percentage between 0 and 100%, inclusive. A permission of 100% means the thread has exclusive access to the location, which in particular means it is allowed to write the location. Any non-zero permission means the thread is allowed to read the location. Our methodology ensures that for each location, the sum of permissions held by the various threads is between 0 and 100%, inclusive; what remains up to 100% is held by the system or by an un-acquired monitor.

Specification of access permissions. To support modular verification, we specify for each method in its precondition the permissions that it requires from its caller, and

```
class Cell {
  int val ;

  Cell Clone()
    requires rd(this.val) ;
    ensures acc(result.val) ∧ rd(this.val) ;
  {
    Cell tmp := new Cell ;
    tmp.val := this.val ;
    return tmp ;
  }
}
```

Fig. 0. A simple example with read and write permissions.

in its postcondition the permissions that it returns to its caller. The *full permission* of 100% for a field f of an object o is denoted by $acc(o.f)$. A *fractional permission* of n% is denoted by $acc(o.f, n)$; that is, $acc(o.f)$ is a shorthand for $acc(o.f, 100)$. Finally, $rd(o.f)$ denotes one *infinitesimal permission* ε, $rd(o.f, k)$ denotes k such permissions, and $rd(o.f, *)$ denotes an inexhaustible supply of ε permissions.

For instance, method $Clone$ in Fig. 0 requires read permission for the location $\mathbf{this}.val$. For a call to $o.Clone()$, the executing thread must possess a non-zero permission for $o.val$. With implicit dynamic frames, frame axioms for methods are expressed implicitly through the specification of access permissions in pre- and postconditions. Instead of providing a separate frame axiom that describes changes of permissions, the evaluation of an assertion changes the permissions. Upon a call to $o.Clone()$, the caller is deprived of the permission required by the precondition, that is, an ε-permission for $o.val$, which is transferred to the callee. Therefore, in the callee method, one may assume that the current thread has at least an ε-permission for $o.val$. However, after the call, the caller may assume this permission only if it is explicitly returned by the method through an appropriate postcondition. This is the case in our example, where the postcondition provides full permission for $\mathbf{result}.val$ and read permission for $\mathbf{this}.val$. If one omitted $rd(\mathbf{this}.val)$ from $Clone$'s postcondition, the caller would not re-gain the permission it had before the call; the executing thread would lose an ε-permission for $o.val$, which would be retained by the system. From then on, no thread could ever obtain full permission for $o.val$, and the location would be immutable.

This form of permission transfer is similar to reasoning in linear logic or capability systems [19]. In particular, the predicate $acc(x) \wedge acc(x)$ is equivalent to false, just like $x \mapsto \_ * x \mapsto \_$ is false in separation logic.

Since method calls change the permissions that may be assumed for the executing thread, it is often useful to think of permissions as being held by method executions rather than by threads. The situation is analogous for loops, where the loop invariant specifies the permissions required and provided by a loop iteration.

Since the evaluation of specifications leads to a transfer of permissions, it must be possible to infer from a specification which permissions to transfer. Therefore, the acc and rd predicates may occur only in positive positions, and not under a quantifier.

Obtaining and using permissions. A thread can obtain permissions in four ways: First, when a thread creates a new object o, it obtains full permission for all fields of o. This exclusive access is justified because o is thread-local until it is explicitly shared with other threads, as we explain below. Second, when a thread acquires the monitor of an object o, it obtains the permissions held by the monitor. The monitor obtained these permissions from the thread that initially shared the object. They are then transferred between the monitor and a thread each time the monitor is acquired or released. Third, when a new thread is forked for an object o, it obtains the permissions required by the precondition of o's Run method. The forking thread is deprived of these permissions. Fourth, when a thread is joined, the joining thread obtains the permissions provided by the postcondition of the Run method of the joined thread, which has then terminated.

Permissions are used to access locations. Each read access to a location $o.f$ generates a proof obligation that the current thread possesses a non-zero permission for $o.f$. Each write access to $o.f$ generates a proof obligation that the current thread possesses full permission for $o.f$.

In method $Clone$ (Fig. 0), the read access to $\mathbf{this}.val$ is permitted because the precondition guarantees that the executing thread has a non-zero permission for this location. The write access to $tmp.val$ is permitted because after tmp's creation, the executing thread has full permission. An attempt to modify $\mathbf{this}.val$ would fail because $Clone$'s precondition does not allow one to prove that the executing thread has full permission for this location.

2 Shared objects

It is possible to share objects between threads. To make a thread-local object available for sharing, the object is first given to the system, which then synchronizes accesses using monitors to ensure a suitable level of mutual exclusion. It is also possible for a shared object to be un-shared, that is, to become thread-local after a period of being shared. In this section, we describe sharing and synchronization, and how they affect access permissions.

Monitors. Objects can be used as *monitors*—locks that protect a set of locations and an invariant [5, 10]. While an object is shared, a thread can acquire it using the **acquire** statement and then release it using the **release** statement. We say that a thread *holds a monitor* if it has acquired, but not yet released the monitor.

The system manages a shared object under a specified *monitor invariant*, declared in the object's class with an **invariant** declaration. Our methodology ensures that the monitor invariant of an object o holds whenever o is shared and o's monitor is not held by any thread. This can be proved by making the monitor invariant a precondition of the share and release operations and a postcondition of the acquire operation.

Like method contracts, monitor invariants specify access permissions along with conditions on variables. For shared objects, these permissions are held by the monitor whenever it is not held by a thread. When a thread acquires the monitor, the permissions are transferred to the acquiring thread, and they are transferred back to the monitor upon release.

```
class $Node$ {
  int $val$ ;
  $Node$ $next$ ;
  int $sum$ ;

  invariant $acc(next) \wedge rd(val)$ ;
  invariant $next \neq \textbf{null} \Rightarrow rd(next.val) \wedge val \leq next.val$ ;
  invariant $acc(sum, 50) \wedge (next = \textbf{null} \Rightarrow sum = 0)$ ;
  invariant $next \neq \textbf{null} \Rightarrow acc(next.sum, 50) \wedge sum = next.val + next.sum$ ;
  invariant $acc(\mu, 50) \wedge (next \neq \textbf{null} \Rightarrow acc(next.\mu, 50) \wedge \mu \sqsubset next.\mu)$ ;
}
```

Fig. 1. Nodes of a sorted linked list.

We illustrate monitor invariants using the linked-list implementation in Fig. 1. Every node of the list stores an integer value, a reference to the next node in the list, and the sum of all values stored in all the successors of the current node. Here, we discuss the first four invariants of class $Node$; the fifth invariant has to do with sharing and the locking order and is discussed later.

The first monitor invariant expresses that the monitor possesses full permission for $\mathtt{this}.next$ and read permission for $\mathtt{this}.val$. (We omit the receiver $\mathtt{this}$ in programs and when it is clear from the context.) Consequently, when a thread acquires the monitor of a node n, it may read and write $n.next$ and read $n.val$. Having at least read permission for these locations allows them to be mentioned in the monitor invariant. For instance, the second invariant states that if there is a successor node, then the present monitor also has read permission for the val field of the successor and that the two nodes are sorted according to their values.

It is important to understand that the monitor invariant of an object o may depend on a location $x.f$ only if o's monitor has (at least) read permission for $x.f$. If this is not the case, the invariant is rejected by the verifier. This requirement is necessary for soundness. For instance, if the invariant of $Node$ did not require $rd(val)$, then it might be possible for some thread to obtain full permission for $n.val$ without acquiring n's monitor. The full permission could then be used to modify $n.val$ and break n's second invariant. When n's monitor is later acquired by another thread, that thread would assume the invariant even though it does not hold, which is unsound.

The third invariant expresses that the monitor holds a fractional permission of 50% for $\mathtt{this}.sum$. Therefore, the invariant is allowed to depend on this location. The fourth invariant states that if there is a successor node, then the present monitor also holds a 50%-permission for the successor's sum location and may, thus, depend on it in its invariant. Using 50%-permissions enables a thread to get full permission for $n.sum$ by acquiring the monitors of n and n's predecessor. It is indeed necessary to acquire both monitors before updating this location because a modification potentially affects the (third) monitor invariant of n as well as the (fourth) monitor invariant of n's predecessor. So both invariants must be checked after an update of $n.sum$, which happens when the monitors are released.

```
class List {
  Node head ;    // sentinel node
  int sum ;

  invariant $acc(head) \land head \neq$ null ;
  invariant $rd(head.val) \land head.val = -1$ ;
  invariant $acc(sum, 20) \land acc(head.sum, 50) \land sum = head.sum$ ;
  invariant $rd(\mu) \land acc(head.\mu, 50) \land \mu \sqsubset head.\mu$ ;

  void Init()
    requires $acc(head) \land acc(sum)$ ;
    requires $acc(\mu) \land \mu = \bot$ ;
    ensures $acc(sum, 80) \land sum = 0$ ;
    ensures $rd(\mu) \land$ maxlock $\sqsubset \mu$ ;
  {
    Node t := new Node ;  t.val := $-1$ ;  t.next := null ;  t.sum := 0 ;
    share t between maxlock and  ;
    head := t ;  sum := 0 ;
    share this between maxlock and t ;
  }

  void Insert(int x)
    requires $acc(sum, 80) \land 0 \leq x \land rd(\mu) \land$ maxlock $\sqsubset \mu$ ;
    ensures $acc(sum, 80) \land sum =$ old(sum) $+ x \land rd(\mu) \land$ maxlock $\sqsubset \mu$ ;
  {
    acquire this ;  sum := sum + x ;
    Node p := head ;  acquire p ;  p.sum := p.sum + x ;
    release this ;
    while ($p.next \neq$ null $\land p.next.val < x$)
      invariant $p \neq$ null $\land acc(p.next) \land acc(p.sum, 50) \land acc(p.\mu, 50)$ ;
      invariant $rd(p.val) \land p.val \leq x \land p.held \land$ maxlock $= p.\mu$ ;
      invariant $p.next =$ null $\Rightarrow p.sum = x$ ;
      invariant $p.next \neq$ null $\Rightarrow$
        $rd(p.next.val) \land p.val \leq p.next.val \land acc(p.next.\mu, 50) \land p.\mu \sqsubset p.next.\mu \land$
        $acc(p.next.sum, 50) \land p.sum = p.next.val + p.next.sum + x$ ;
      lockchange p ;
    {
      Node nx := p.next ;  acquire nx ;  nx.sum := nx.sum + x ;
      release p ;  p := nx ;
    }
    Node t := new Node ;  t.val := x ;  t.next := p.next ;
    if (t.next = null) { t.sum := 0 ; } else { t.sum := p.next.val + p.next.sum ; }
    share t between p and p.next ;
    p.next := t ;
    release p ;
  }
}
```

Fig. 2. Main class of the sorted linked list. The **while** statement in method *Insert* includes a loop invariant and a **lockchange** clause that says how a loop iteration may affect what locks the thread holds.

Fig. 2 shows the implementation of the main class of the linked list. According to the third monitor invariant, the monitor of a $List$ object l holds a 20%-permission for $l.sum$, which allows the monitor invariant to depend on the location. Threads may hold parts of the remaining 80% and read the location without acquiring l's monitor. But only a thread that holds exactly 80% can obtain write permission for $l.sum$ by acquiring the monitor. The exact percentages for the fractional permissions here are arbitrary; we could as well have chosen 50% or any other non-zero percentage.

Just like the monitor of a $Node$ object holds a 50%-permission for the sum location of the next node, the monitor of a $List$ object l holds a 50%-permission for the sum location of the first node $l.head$. Therefore, to obtain write permission for $l.head.sum$, a thread has to acquire not only the monitor of $l.head$ but also the monitor of l, which protects $List$'s third monitor invariant.

Method $Insert$ of class $List$ inserts a new value into the list. It uses fine-grained hand-over-hand locking to traverse the list. This locking strategy ensures that once the method finds the appropriate place to insert the new element, it holds the lock of the new node's predecessor. Moreover, it enables us to update the sum field while traversing the list. Hand-over-hand locking becomes possible by our use of fractional permissions in the monitor invariant of $Node$.

Sharing and unsharing. Every object is either thread-local or shared. An object is thread-local upon creation. A thread-local object o is shared by the **share** o statement; conversely, a shared object o is made thread-local by the **unshare** o statement.

Sharing an object o transfers the permissions required by o's monitor invariant from the current thread to o's monitor. That is, the **share** o statement checks that o is a thread-local object, after which it makes o shared. It then checks that o's monitor invariant holds, in particular, that the current thread holds all the permissions required by o's monitor invariant. Finally, it deprives the current thread of these permissions.

Conversely, **unshare** o checks that o is a shared object, after which it makes o thread-local. Whereas **share** o requires o to be thread-local, which implies its monitor is not held by any thread, **unshare** o requires o's monitor to be held by the current thread and then releases the monitor of o.

Note that **unshare** o does not necessarily give the current thread full permissions for o's fields. The thread obtains only the permissions held by o's monitor, but other threads might still hold permissions. Therefore, thread-locality of o means only that no thread can acquire o's monitor, but other threads might still access o's fields.

Method $Init$ of class $List$ (Fig. 2) illustrates sharing. The method plays the role of a constructor, that is, it is expected to be called on newly allocated $List$ objects. Hence, it requires write permissions for the $head$ and sum fields of its receiver. The second precondition requires that the receiver be thread-local, as we discuss later. The method creates and initializes a new $Node$ object t. Since t is thread-local and since $t.next$ is null, the current thread possesses all the permissions required by t's monitor invariant (Fig. 1). Therefore, the **share** t statement verifies (we will explain the **between** clause in the next section). The **share this** statement verifies because the current thread possesses all the permissions required by the monitor invariant of **this**. In particular, when t is being shared, the current thread retains a read permission for $t.val$

(since *Node*'s monitor invariant requires only an ε-permission) and a 50%-permission for $t.sum$ (since *Node* requires only a 50%-permission). *Init* satisfies its first postcondition because the current thread retains an 80%-permission for $\mathbf{this}.sum$ when **this** is being shared (since *List*'s invariant requires only 20%), and because sum is set to zero by the method. We will discuss the second postcondition in the next section.

It is interesting to trace the permissions for $t.val$. After creating t, the current thread possesses full permission for this location. Sharing t transfers an ε-permission to t's monitor, such that the current thread retains $100\% - \varepsilon$. Consequently, by acquiring t's monitor, the thread could now re-gain write permission for $t.val$. Later, when **this** is being shared, another ε-permission is transferred to the monitor of **this**, which leaves the current thread with $100\% - 2 \cdot \varepsilon$. However, when the *Init* method terminates, this remaining permission is not transferred to the caller. Therefore, it is effectively lost for all threads, and $t.val$ is from then on immutable.

3 Deadlock prevention

To prevent deadlocks, locks must be acquired in ascending order, according to a user-defined locking order. In this section, we show how this order is defined, how it is enforced, and how programs can set and change an object's position in the order.

Locking order. To allow the locking order to be changed dynamically, we store each object's position in a predefined field μ. The type of μ is a lattice in which for any two distinct, ordered elements u and v, there is some element w strictly in between them. This requirement ensures that it is always possible to place an object between any two existing objects in the locking order. We use $u \sqsubset v$ to denote that u is strictly less than v in the lattice. The bottom element of the lattice is denoted by $\bot$.

As for other fields, accesses to μ require the appropriate permissions. However, μ may be modified only through the **share** statement and the **reorder** statement described below. The μ field may be used in specifications. For instance, the last invariant of class *Node* (Fig. 1) specifies the locking order between a node and its successor. To do so, it requires 50%-permissions for the μ fields of both nodes and orders **this** before its successor. Consequently, the monitors of the nodes have to be acquired in the order of the nodes in the list. Similarly, the last invariant of *List* (Fig. 2) orders **this** before the first node; so the *List* object must be acquired before its nodes.

We use the μ field also to encode whether an object is thread-local or shared. An object o is thread-local if and only if $o.\mu = \bot$. For instance, the second precondition of method *Init* (Fig. 2) requires that the receiver be thread-local, and the second postcondition ensures that it is shared (since for all u, $u \sqsubset \mu$ implies $o.\mu \neq \bot$).

Acquiring monitors. To check that a thread acquires monitors in the specified order, we have to keep track of the monitors held by each thread. We use the expression $\mathbf{maxlock} \sqsubset u$ to express that u is greater than $o.\mu$ for each object o currently locked by the current thread. Since this expression implicitly reads $o.\mu$ for all objects held by the current thread, we ensure that $o.\mu$ may be changed only by the thread that holds o, see below.

The proof obligation for $\mathbf{acquire}\ o$ ensures that monitors are acquired in ascending order, that is, that the current thread has read permission for $o.\mu$ and that o is strictly above all objects already held by the current thread. Note that because of this proof obligation, it is (allowed but) not sensible to require full permission for $\mathbf{this}.\mu$ in the monitor invariant of an object o: when o is being shared, its monitor would obtain full permission to $o.\mu$; so no thread could possess read permission for $o.\mu$ and, thus, no thread could ever acquire o's monitor.

Determining the locking order. The locking order is specified and changed by the $\mathbf{between}\ \overline{p}\ \mathbf{and}\ \overline{s}$ clause of the $\mathbf{share}\ o$ and $\mathbf{reorder}\ o$ statements, for any (possibly empty) lists of expressions $\overline{p}$ and $\overline{s}$. It assigns a value to $o.\mu$ that is strictly above all the lower bounds $p_i.\mu$ and strictly below all the upper bounds $s_j.\mu$. The operations require the current thread to have write permission for $o.\mu$ and read permission for all $p_i.\mu$ and $s_j.\mu$ and require each lower bound $p_i.\mu$ to be strictly below each upper bound $s_j.\mu$. Whereas the $\mathbf{share}$ statement places a thread-local object in the locking order, $\mathbf{reorder}\ o$ is used to change the position of a shared object o, which must be held by the current thread to prevent one thread from confusing another thread's $\mathbf{maxlock}$ value.

In $List$'s $Init$ method (Fig. 2), the new (thread-local) $Node$ object t is ordered above $\mathbf{maxlock}$, which lets $\mathbf{share\ this}$ order the (thread-local) $\mathbf{this}$ object between $\mathbf{maxlock}$ and t, as required by the last postcondition of $Init$ and $List$'s last monitor invariant, respectively. Since we are not interested in ordering t below any particular object, the second expression list of the $\mathbf{share}\ t$ statement is empty.

It is an important feature of our verification methodology that the μ field of an object can be assigned to more than once, that is, the locking order can be changed during program execution. In our example, the monitor invariant of $Node$ (Fig. 1) requires 50%-permissions for $\mathbf{this}.\mu$ and $next.\mu$. Therefore, it is possible for a thread to acquire the monitors of nodes n and $n.next$, and thus, obtain full permission for $n.next.\mu$. Consequently, the thread can change the place of $n.next$ in the locking order. We used this feature to implement an association list that re-orders its nodes after each lookup to ensure that frequently-used elements appear toward the head of the list. List reversal and balanced trees are other common examples that require a dynamic change of the locking order.

4 Technical treatment

In this section, we explain how our methodology is encoded in the program verifier. We define the proof rules for the most interesting statements by translating them to a simple guarded-command language, whose weakest precondition semantics is obvious. In this translation, we use $\mathbf{assert}$ statements to denote proof obligations and $\mathbf{assume}$ statements to state assumptions that may be used to prove the assertions. The heap is encoded as a two-dimensional array that maps objects and field names to values. The current heap is denoted by the global variable $Heap$.

Encoding of permissions. A permission has the form (p, n) where p is a percentage between 0 and 100, and n is either an integer or one of the special values $-\infty$ or $+\infty$. These special values are used to represent an inexhaustible supply of ε permissions, as expressed by the predicate $rd(o.f, *)$. We use integral percentages rather than the mathematically more appealing fractions, due to a limitation in many popular SMT solvers in their handling of both integers and rationals. Intuitively, we define the value of a permission (p, n) as $p + n \cdot \varepsilon$, where ε is a positive infinitesimal.

Percentages are a simple way to encode fractions of a definite size, which are for instance needed to split permissions over a statically-known number of monitors or threads. Infinitesimals allow one to split permissions between arbitrarily many monitors and threads, for instance, to allow a statically-unknown number of concurrent readers.

A permission (p, n) is called:

- full permission if $p + n \cdot \varepsilon = 100$, that is, $p = 100 \wedge n = 0$;
- some permission if $p + n \cdot \varepsilon > 0$, that is, $p > 0 \vee n > 0$;
- no permission if $p + n \cdot \varepsilon = 0$, that is, $p = 0 \wedge n = 0$.

Other combinations of p and n do not occur. Note that our encoding does not reflect that ε is an infinitesimal. It simply counts the number of such ε's (or "tokens").

We assume the following operations on permissions: incrementing (denoted by $+$) and decrementing (denoted by $-$) by a percentage or by a possibly inexhaustible number of infinitesimal permissions ε, and comparison ($=$, $<$, $\leq$). The definitions of these operations are straightforward and, therefore, omitted.

To keep track of the permissions it holds, each thread t has a (thread-local) variable $\mathcal{P}_t$ that maps every location to t's permission for that location. Since specifications are given with respect to one thread (the current thread, denoted by tid) and, likewise, verification conditions are prescribed for each thread, we usually refer only to one variable $\mathcal{P}_{tid}$, so we drop the subscript tid.

It is convenient to introduce shorthands for the two most common permission requirements. $CanRead(o.f)$ and $CanWrite(o.f)$ express that the current thread holds some permission and full permission for location $o.f$, respectively:

$$\begin{aligned} CanRead(o.f) &\equiv o \neq \mathbf{null} \wedge \mathbf{let}\ (p, n) = \mathcal{P}_{tid}[o,f]\ \mathbf{in}\ p > 0 \vee n > 0 \\ CanWrite(o.f) &\equiv o \neq \mathbf{null} \wedge \mathbf{let}\ (p, n) = \mathcal{P}_{tid}[o,f]\ \mathbf{in}\ p = 100 \wedge n = 0 \end{aligned}$$

Object creation. For any class C and local variable x, the allocation statement is given the following semantics:

```
$x := \mathbf{new}\ C;\quad \equiv$
  $\mathbf{havoc}\ x;$
  $\mathbf{assume}\ x \neq null \wedge (\forall f \bullet \mathcal{P}[x,f] = (0,0) \wedge Heap[x,f] = zero\ );$
  $\#\mathbf{foreach}\ f\ \{\ \mathcal{P}[x,f] := (100, 0);\ \}$
```

The $\mathbf{havoc}\ x$ statement assigns an arbitrary value to x, which is then constrained by the following $\mathbf{assume}$ statement. $zero$ denotes the zero-equivalent value for each type, in particular, $\perp$ for the locking order. Note that this semantics is simplified. In particular, we do not express here that the new object is an instance of class C or that the f in the $\#\mathbf{foreach}$ statement is a field of class C, because these are not relevant for our discussion. $\#\mathbf{foreach}$ loops can be statically expanded by the translator.

Field access. Reading and writing locations first checks that the thread has the appropriate permission:

```
$x := o.f;\ \ \equiv$                              $o.f := x;\ \ \equiv$
    assert $CanRead(o.f)$;                          assert $CanWrite(o.f)$;
    $x := Heap[o, f]$;                              $Heap[o, f] := x$;
```

Monitors. Each thread keeps track of the monitors it holds. For that purpose, we introduce a thread-local boolean field $held$. As with $\mathcal{P}$, this field would be subscripted with the thread, but since we only refer to the field for the current thread, we drop the subscripts. That is, $Heap[o, held]$ denotes whether the monitor of object o is held by the current thread. Since $held$ is thread-local, it is not subject to permission checks; each thread always has full permission for its $held$ fields.

The expression **maxlock** is encoded using quantifiers over the objects whose monitors are held by the current thread. For instance, **maxlock** $\sqsubset u$ is encoded as $(\forall p \bullet Heap[p, held] \Rightarrow Heap[p, \mu] \sqsubset u)$.

Permission transfer. Several statements of our programming language transfer permissions between threads and monitors (for instance, **acquire**), two threads (for instance, **fork**, see below), or between two method executions of the same thread (method call). We model this permission transfer by two operations, Exhale and Inhale, which describe the transfer from the current thread's perspective.

Roughly speaking, Exhale$[\![E]\!]$ checks that expression E holds, in particular, that the current thread holds the permissions required by E, and then takes away these permissions. Inhale$[\![E]\!]$ assumes E and transfers the permissions required by E to the current thread. If the current thread obtains some permission for a location $o.f$ for which it previously had no permission, Inhale assigns an arbitrary value to $o.f$, which models the fact that another thread might have modified the location since the current thread last accessed it. The definitions for both operations are shown in Fig. 3.

Acquiring and releasing monitors. The precondition of **acquire** o requires object o to be ordered above all objects already held by the acquiring thread. This proof obligation also ensures that o is shared, because our encoding is consistent with a model where every thread holds an anonymous sentinel monitor; in particular, it is not possible to refute $Heap[\perp, held] = true$. To ensure mutual exclusion, the execution of the **acquire** statement suspends until no other thread holds o's monitor. The Inhale operations expresses that the acquiring thread may assume the monitor invariant of o, denoted by $J(o)$, and that it obtains the permissions held by o's monitor.

The **release** o statement requires o's monitor to be held by the current thread. Using the Exhale operation, it then asserts o's monitor invariant and transfers permissions back to the monitor:

```
acquire $o$;  $\equiv$                                                          release $o$;  $\equiv$
  assert $CanRead(o.\mu)$;                                                 assert $o \neq null$;
  assert $(\forall p \bullet Heap[p, held] \Rightarrow Heap[p, \mu] \sqsubset Heap[o, \mu])$;   assert $Heap[o, held]$;
  $Heap[o, held] :=$ true;                                                 Exhale$[\![J(o)]\!]$
  Inhale$[\![J(o)]\!]$                                                       $Heap[o, held] :=$ false;
```

```
Exhale[[$acc(E.f, r)$]]  $\equiv$
   assert $\mathcal{P}[\mathsf{Tr}[\![E]\!], f] \geq \mathsf{Tr}[\![r]\!]$;
   $\mathcal{P}[\mathsf{Tr}[\![E]\!], f] := \mathcal{P}[\mathsf{Tr}[\![E]\!], f] - \mathsf{Tr}[\![r]\!]$;

Exhale[[$rd(E.f)$]]  $\equiv$
   assert $\mathcal{P}[\mathsf{Tr}[\![E]\!], f] \geq \varepsilon$;
   $\mathcal{P}[\mathsf{Tr}[\![E]\!], f] := \mathcal{P}[\mathsf{Tr}[\![E]\!], f] - \varepsilon$;

Exhale[[$P \wedge Q$]]  $\equiv$
   Exhale[[$Q$]];
   Exhale[[$P$]];

Exhale[[$P \Rightarrow Q$]]  $\equiv$
   if (Tr[[$P$]]) { Exhale[[$Q$]]; }

Otherwise:

Exhale[[$E$]]  $\equiv$
   assert Tr[[$E$]];
```

```
Inhale[[$acc(E.f, r)$]]  $\equiv$
   if ($\mathcal{P}[\mathsf{Tr}[\![E]\!], f] = (0, 0)$)
      havoc $Heap[\mathsf{Tr}[\![E]\!], f]$;
   $\mathcal{P}[\mathsf{Tr}[\![E]\!], f] := \mathcal{P}[\mathsf{Tr}[\![E]\!], f] + \mathsf{Tr}[\![r]\!]$;

Inhale[[$rd(E.f)$]]  $\equiv$
   if ($\mathcal{P}[\mathsf{Tr}[\![E]\!], f] = (0, 0)$)
      { havoc $Heap[\mathsf{Tr}[\![E]\!], f]$; }
   $\mathcal{P}[\mathsf{Tr}[\![E]\!], f] := \mathcal{P}[\mathsf{Tr}[\![E]\!], f] + \varepsilon$;

Inhale[[$P \wedge Q$]]  $\equiv$
   Inhale[[$P$]];
   Inhale[[$Q$]];

Inhale[[$P \Rightarrow Q$]]  $\equiv$
   if (Tr[[$P$]]) { Inhale[[$Q$]]; }

Otherwise:

Inhale[[$E$]]  $\equiv$
   assume Tr[[$E$]];
```

Fig. 3. Exhale$[\![E]\!]$ and Inhale$[\![E]\!]$ are defined by structural induction over expression E. The function Tr translates source expressions to our intermediate language. We assume here that acc and rd expressions only occur on the outermost level of conjuncts and consequences of implications. Therefore, Tr never encounters these expressions. Exhale$[\![E]\!]$ also asserts that E is well-defined, in particular, that the current thread possesses the permissions needed for the field accesses in E. We omit these checks and related technicalities for simplicity.

Finally, the **reorder** statement requires write permission for $o.\mu$, that o is held by the current thread, and that any lower bound $p_i.\mu$ is below any upper bound $s_j.\mu$. It then chooses an appropriate value w for $o.\mu$ and assigns it. Recall from Section 3 that the lattice of positions in the locking order guarantees that for any two distinct, ordered elements u and v, there is some element w strictly in between them. Therefore, it is always possible to choose an appropriate value for $o.\mu$:

```
reorder $o$ between $\overline{p}$ and $\overline{s}$;   $\equiv$
   assert $CanWrite(o, \mu) \wedge Heap[o, held]$;                                    ⎫
   #foreach $p_i \in \overline{p}, s_j \in \overline{s}$ {                                        ⎪
      assert $p_i = null \vee s_j = null \vee$                                         ⎪
            $(CanRead(p_i.\mu) \wedge CanRead(s_j.\mu) \wedge Heap[p_i, \mu] \sqsubset Heap[s_j, \mu])$;  ⎪
   }                                                                                   ⎬ (*)
   havoc $w$;                                                                          ⎪
   #foreach $p_i \in \overline{p}$ { assume $p_i = null \vee Heap[p_i, \mu] \sqsubset w$; } ;   ⎪
   #foreach $s_j \in \overline{s}$ { assume $s_j = null \vee w \sqsubset Heap[s_j, \mu]$; } ;   ⎪
   $Heap[o, \mu] := w$;                                                                ⎭
```

Sharing and unsharing. An object o can be shared if the current thread has write permission for $o.\mu$ and if the object is not shared already. Like for $\mathbf{release}$, the Exhale operation is used to check that the current thread has the permissions required by o's monitor invariant $J(o)$ and transfers them to the monitor.

The $\mathbf{unshare}$ o statement releases o and at the same time makes it unavailable for sharing by setting $o.\mu$ to $\bot$. Since it changes $o.\mu$, the $\mathbf{unshare}$ statement requires full permission for $o.\mu$. This requirement also ensures mutual exclusion with the $\mathbf{acquire}$ o statement, which requires read permission for $o.\mu$.

```
share $o$ between $\overline{p}$ and $\overline{s}$;  $\equiv$
    assert $CanWrite(o, \mu)$;
    assert $Heap[o, \mu] = \bot$;
    // see $(*)$ of reorder
    Exhale$[\![J(o)]\!]$
```

```
unshare $o$;  $\equiv$
    assert $CanWrite(o, \mu)$;
    assert $Heap[o, held]$;
    $Heap[o, held] :=$ false;
    $Heap[o, \mu] := \bot$;
```

Thread creation and termination. Every object o can give rise to a computation, which is performed in a separate thread as if, in Java, every object were an instance of class *Thread*. The **fork** o statement starts such a computation by executing o's *Run* method. Like in Java, we do not permit several overlapping computations on the same object, which allows us in particular to identify a thread through the object on which it was forked. To prevent overlaps, we introduce a boolean field *active* to record whether there is an active computation on an object. For new objects, *active* is initially false. The **fork** o statement asserts that the current thread has write permission for $o.active$ and that o is not active. It also asserts the precondition of o's *Run* method, denoted by $RunPre(o)$, and transfers the required permissions to the new thread using the Exhale operation. The new thread will then execute o's *Run* method.

The **join** o statement waits for the computation of the thread that has been forked on object o to complete, and then marks o as no longer being active. The current thread may assume the postcondition of o's *Run* method, denoted by $RunPost(o)$, and obtains the permissions of the joined thread.

```
fork $o$;  $\equiv$
    assert $CanWrite(o.active)$;
    assert $\neg Heap[o, active]$;
    Exhale$[\![RunPre(o)]\!]$
    $Heap[o, active] := true$;
```

```
join $o$;  $\equiv$
    assert $CanWrite(o.active)$;
    assert $Heap[o, active]$;
    $Heap[o, active] := false$;
    Inhale$[\![RunPost(o)]\!]$
```

Note that requiring write permission for $o.active$ in both **fork** o and **join** o ensures mutual exclusion. In particular, a thread can be joined only once, which prevents a duplication of the permissions returned from that thread.

When the *Run* method is initiated by a **fork**, then its specification is interpreted from two different threads: the precondition is exhaled by the forking thread and inhaled by the forked thread; the postcondition is exhaled by the terminating thread and inhaled by the joining thread. Therefore, it is necessary for soundness that these interpretations are consistent. We achieve that by restricting the use of thread-local fields. The specification of *Run* must not mention the *held* field of any object. Moreover, since **maxlock** is encoded in terms of *held*, it may be used only in the form $\mathbf{maxlock} \sqsubset E$ in positive contexts of the precondition.

Method calls and loops. The semantics of method calls exhales the precondition and then inhales the postcondition. In this way, it is like the succession of a fork and a join, except for the $active$ machinery, and without the restrictions on the specification of the Run method. Indeed, fork and join are nothing but an asynchronous call to a method called Run.

The **while** statement exhales the loop invariant and then havocs the variables assigned to in the loop body. Then, it either inhales the loop invariant, assumes the negation of the loop guard, and continues after the loop, or it starts from an empty mask $\mathcal{P}$, inhales the loop invariant, assumes the loop guard, executes the loop body, and exhales the loop invariant. For brevity, we omit the formalization.

5 Related Work

Implicit dynamic frames were first used by Smans *et al.* [18] as a way to use Kassios's dynamic frames [15] but with access predicates instead of explicit modifies clauses. The permissions required by a method precondition implicitly define an access set, which is an upper bound on the fields modified by the method. We extend this work by supporting fractional permissions, which call for the exhale and inhale operations instead of just computing access sets. The havoc in the inhale operation corresponds to the havoc of the heap in the encoding of a modifies clause.

Fractional permissions were proposed by Boyland [4] and used by Zhao [21] for the analysis of concurrent Java programs. Zhao developed a type system to track read and write permissions for fields and to enforce a (fixed) locking order. The type system enforces the absence of data races and deadlocks, but does not support the verification of a program w.r.t. to a programmer-supplied contract.

Methodologies similar to ours have been defined in separation logic by Bornat *et al.* [2], by Gotsman *et al.* [8], and by Hobor *et al.* [11]. These extend Concurrent Separation Logic [17] to allow an unbounded number of locks and threads and to allow fractional permissions and counting permissions, which are similar to our infinitesimal permissions. A difference is that we translate our methodology into first-order verification conditions instead of needing a separate logic. A minor difference with separation logic is that we can handle **old** expressions, which provide a natural way to write postconditions. Unlike these pieces of work, we also verify that programs do not have deadlocks.

Checkers for separation logic include Smallfoot [1], jStar [7], and VeriFast [12], which are all based on some symbolic execution with interspersed calls to a theorem prover. By translating each method to just one formula, we can let the theorem prover perform case splits that a symbolic execution engine would have to resolve at each program point, which is not always possible. On the other hand, we currently have no support for abstract predicates and currently do not check that permissions are not lost.

Boyapati *et al.* [3] present an ownership type system that prevents data races and deadlocks. This system supports thread-local objects and coarse-grained locking of shared objects, where the lock of an object o also protects all objects owned by o. The type system permits concurrent reading only for immutable objects, whereas the fractional permissions in our system support fine-grained locking and concurrent reading. Similar to our work, Boyapati *et al.*'s system prevents deadlocks by enforcing that locks are acquired in a given locking order, and this order can be changed dynamically.

Jacobs *et al.* [13] extend Spec#'s verification methodology to concurrent programs. Like Boyapati, they use ownership to impose a coarse-grained locking strategy, whereas our methodology supports fine-grained locking of arbitrary structures. We adopted their technique of specifying the locking order as part of the **share** statement and extended this capability by allowing locks to be re-ordered.

6 Conclusions

We presented a verification methodology for concurrent, object-based programs, which enforces the absence of data races and deadlocks and allows one to verify code against contracts. Our methodology uses fractional permissions, which allow us to support fine-grained locking and multi-object monitor invariants, sharing and un-sharing of objects, and concurrent reading. Our methodology encodes the locking order via fields in the heap, which enables dynamic changes. These features make our methodology sufficiently expressive to verify advanced concurrency patterns.

We have implemented our methodology in a translator from our experimental source language Chalice to the intermediate verification language Boogie [0] and have used it to verify several challenging examples including hand-over-hand locking for linked lists and a lock re-ordering algorithm. We have designed our methodology to work well with off-the-shelf SMT solvers and, indeed, all of our examples could be verified fully automatically. Our implementation also supports reader-writer locks, which we omitted here for lack of space.

The presented methodology is an expressive foundation for more comprehensive verification techniques. As future work, we plan to prove a formal soundness result including the following properties: (0) Justification of assumptions: the conditions assumed as part of the Inhale operation are guaranteed to hold. (1) Non-interference of threads, in particular, stability of read expressions: no thread can be writing an expression that is being read by another thread. (2) Absence of deadlocks in the presence of our changing locking order. Other plans for future work are to extend our methodology by two-state invariants to permit rely-guarantee reasoning and by abstraction via user-defined functions or predicates. We also want to develop an automatic inference of access predicates and extend Chalice to a full object-oriented language by adding subtyping.

Acknowledgments. We thank the referees for their useful comments. Müller's work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

0. 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 *FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.
1. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO 2005*, volume 4111 of *LNCS*, pages 115–137. Springer, 2006.
2. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL 2005*, pages 259–270. ACM, 2005.
3. C. Boyapati, R. Lee, and M. C. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA 2002*, pages 211–230. ACM, 2002.
4. J. Boyland. Checking interference with fractional permissions. In *SAS 2003*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.
5. P. Brinch Hansen. *Operating systems principles*. Prentice-Hall, Englewood Cliffs, NJ, 1973.
6. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
7. D. Distefano and M. J. Parkinson. jStar: Towards practical verification of Java. In *OOPSLA 2008*, pages 213–226. ACM, 2008.
8. A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *APLAS 2007*, volume 4807 of *LNCS*, pages 19–37. Springer, 2007.
9. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
10. C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, Oct. 1974.
11. A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle semantics for concurrent separation logic. In *ESOP 2008*, volume 4960 of *LNCS*, pages 353–367. Springer, 2008.
12. B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW520, Katholieke Universiteit Leuven, Aug. 2008.
13. B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A statically verifiable programming model for concurrent object-oriented programs. In *ICFEM 2006*, volume 4260 of *LNCS*, pages 420–439. Springer, 2006.
14. C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332. North-Holland, 1983.
15. I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM 2006*, volume 4085 of *LNCS*, pages 268–283. Springer, 2006.
16. D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 1999.
17. P. W. O'Hearn. Resources, concurrency, and local reasoning. *TCS*, 375(1–3):271–307, 2007.
18. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. In *FTfJP 2008*, pages 1–12, 2008. Technical Report ICIS-R08013, Radboud University.
19. D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM TOPLAS*, 22(4):701–771, 2000.
20. Q. Xu, W.-P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.
21. Y. Zhao. *Concurrency Analysis based on Fractional Permission System*. PhD thesis, The University of Wisconsin–Milwaukee, 2007.