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

Manuscript KRML 191, DRAFT, 9 October 2008.

Abstract. Advanced multi-threaded programs apply concurrency concepts in a sophisticated way. 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

Maintream concurrent programs use multiple threads and synchonization 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 [12, 7].

The above patterns improve the performance and flexibility of programs, but they also complicate reasoning. For instance, a typical consequence of fine-grained locking is that several locks need to 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 $next \neq \mathbf{null} \Rightarrow val \leq next.val$. 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 [10, 15] 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 [13] 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 [2], 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 [1, 6], but our approach generates verification conditions in first-order logic, well-suited for off-the-shelf SMT solvers such as Simplify [5] and Z3 [4]. Finally, our methodology permits not a fixed but a changeable locking order among monitors. This allows lock re-ordering, which as far as we know makes our static verifier the first to incorporate such a feature. 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. The appendix contains additional examples that highlight interesting aspects of our methodology.

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 permis-

```
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.

sions 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 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 calling thread must possess any 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 passed 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 passing is similar to reasoning in linear logic or capability systems [14]. 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.

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 locations 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 passed 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 $o.f$ is guarded by a proof obligation that the current thread possesses a non-zero permission for $o.f$. Each write access to $o.f$ is guarded by 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 [3, 8]. 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. 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) ∧ rd(val) ;
  invariant next ≠ null ⇒ rd(next.val) ∧ val ≤ next.val ;
  invariant acc(sum, 50) ∧ (next = null ⇒ sum = 0) ;
  invariant next ≠ null ⇒ acc(next.sum, 50) ∧ sum = next.val + next.sum ;
  invariant next ≠ null ⇒ acc(next.shared) ∧ next.shared ;
  invariant acc(μ, 50) ∧ (next ≠ null ⇒ acc(next.μ, 50) ∧ μ ⊏ next.μ) ;
}
```

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 remaining invariants have to do with sharing and the locking order, and are discussed later.

The first monitor invariant expresses that the monitor possesses full permission for $\mathbf{this}.next$ and read permission for $\mathbf{this}.val$. (We omit the receiver $\mathbf{this}$ in programs and when it is clear form 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 mention 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 $\mathbf{this}.sum$. Therefore, the monitor invariant is allowed to read this location, as it is the case in the second conjunct of this invariant. 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, read 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) ∧ head ≠ null ;
  invariant rd(head.val) ∧ head.val = −1 ;
  invariant acc(sum, 20) ∧ acc(head.sum, 50) ∧ sum = head.sum ;
  invariant acc(head.shared) ∧ head.shared ;
  invariant rd(μ) ∧ acc(head.μ, 50) ∧ μ ⊏ head.μ ;

  void Init()
    requires acc(shared) ∧ ¬shared ∧ acc(head) ∧ acc(sum) ;
    requires acc(μ) ;
    ensures acc(shared) ∧ shared ∧ acc(sum, 80) ∧ sum = 0 ;
    ensures rd(μ) ∧ maxlock ⊏ μ ;
  {
    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 ;
  }

  // see appendix for Insert method
}
```

Fig. 2. Main class of the sorted linked list.

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 read 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 a write permission to $l.head.sum$, a thread has to acquire the monitors of l and $l.head$, which protects $List$'s third monitor invariant.

In App. A.0, we present the $Insert$ method of class $List$, which uses fine-grained hand-over-hand locking to traverse the list.

Sharing and unsharing. Every object contains a predefined boolean field $shared$, which indicates whether an object is shared or thread-local. Since fresh objects are thread-local after their creation, $shared$ is initially false. 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.

Access to $shared$ is governed by the same permission rules as for any other field. However, $shared$ can be changed only through the **share** and **unshare** statements. The $shared$ field may also be used in monitor invariants. For instance, the fifth invariant of $Node$ and the fourth invariant of $List$ ensure that all nodes of a list are shared.

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 the current thread has write permission for $o.shared$ and that o is a thread-local object, after which it changes $o.shared$ from false to true. 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 the current thread has write permission for $o.shared$ and that o is a shared object, after which it changes $o.shared$ back from true to false. 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. The **unshare** o statement ends by releasing the monitor of o.

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 fresh $List$ objects. Therefore, it requires write permissions for the $head$, sum, and $shared$ locations of its thread-local receiver. 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), a 50%-permission for $t.sum$ (since $Node$ requires only a 50%-permission), and full permission for $t.shared$ (since $Node$ does not require any permission for this location). $Init$ satisfies its first postcondition because (0) the current thread retains full permission for **this**$.shared$ when **this** is being shared, (1) $shared$ is set to true by the **share this** statement, (2) the current thread retains an 80%-permission for **this**$.sum$ when **this** is being shared (since $List$'s invariant requires only 20%), and (3) 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 passes 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 passed 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 passed to the caller. Therefore, it is effectively lost for all threads, and $t.val$ is from then on immutable.

3 Deadlock prevention

In order to prevent deadlocks, locks must be acquired in ascending order, according to a user-defined locking order. In this section, we show how the locking order is defined,

how it is enforced, and how programs can set and change an object's position in the locking 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 p and s, there is some element t 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 $t \sqsubset s$ to denote that t is strictly less than s in the lattice.

Accesses to μ require the appropriate permissions. However, μ may be modified only through the $\mathbf{share}$ statement and the $\mathbf{install}$ statement described below. Like any other field, μ 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 locations of both nodes and orders $\mathbf{this}$ 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 $\mathbf{this}$ before the first node; so the $List$ object must be acquired before its nodes can be acquired.

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 $\mathbf{maxlock} \sqsubset p$ to express that p is greater than $o.\mu$ for each object o currently locked by the current thread. Since this expression implicitly reads the μ field of all objects locked by the current thread, we ensure that each thread holds at least read permission for all objects whose monitor it holds; we describe this mechanism in Section 4.

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 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 $\mathbf{install}\ o$ 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 operation requires the current thread to have write permission for $o.\mu$ and read permission for all $p_i.\mu$ and $s_j.\mu$ and requires each lower bound $p_i.\mu$ to be strictly below each upper bound $s_j.\mu$.

In $List$'s $Init$ method (Fig. 2), the new $Node$ object t is ordered above $\mathbf{maxlock}$, which lets $\mathbf{share}\ \mathbf{this}$ order $\mathbf{this}$ 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 $Node$ 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, see App. A.1. List reversal is another common example that requires 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 $+\infty$. Intuitively, we define the value of a permission (p, n) as $p + n \cdot \varepsilon$, where ε is a positive infinitesimal. 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, $p = 0 \ \wedge \ n = 0$.

All other combinations of p and n do not occur. We use percentages rather than fractions to keep our input language simple.

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:

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

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 (0, $\mathbf{false}$, $\mathbf{null}$). 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 is the $\mathbf{foreach}$ statement is a field of class C, because these are not relevant for our discussion.

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

$x := o.f; \quad \equiv$
  $\mathbf{assert}\ \mathit{CanRead}(o.f);$
  $x := \mathit{Heap}[o,f];$

$o.f := x; \quad \equiv$
  $\mathbf{assert}\ \mathit{CanWrite}(o.f);$
  $\mathit{Heap}[o,f] := x;$

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, $\mathit{Heap}[o, \mathit{held}]$ denotes whether the monitor of object o has been acquired 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 $\mathbf{maxlock}$ is encoded using quantifiers over the objects whose monitors are held by the current thread. For instance, $\mathbf{maxlock} \sqsubset s$ is encoded as $(\forall p \bullet \mathit{Heap}[p, \mathit{held}] \Rightarrow \mathit{Heap}[p, \mu] \sqsubset s\ )$.

Permission transfer. Several statements of our programming language transfer permissions between threads and monitors (for instance, $\mathbf{acquire}$), two threads (for instance, $\mathbf{fork}$, see below), or between two method incarnations 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 $o.f$. This $\mathbf{havoc}$ 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 $\mathbf{acquire}\ o$ requires that object o is shared and that the acquiring thread acquires locks in ascending order. To ensure mutual exclusion, the execution of the $\mathbf{acquire}$ statement suspends until no other

$\mathsf{Exhale}[\![\mathbf{acc}(E.f, r)]\!] \equiv$
  if $(f = \mu \wedge \mathit{Heap}[\mathsf{Tr}[\![E]\!], \mathit{held}])$
    assert $\mathcal{P}[\mathsf{Tr}[\![E]\!], f] > \mathsf{Tr}[\![r]\!]$;
  else
    assert $\mathcal{P}[\mathsf{Tr}[\![E]\!], f] \geq \mathsf{Tr}[\![r]\!]$;
  $\mathcal{P}[\mathsf{Tr}[\![E]\!], f] := \mathcal{P}[\mathsf{Tr}[\![E]\!], f] - \mathsf{Tr}[\![r]\!]$;

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

$\mathsf{Exhale}[\![\mathbf{rd}(E.f)]\!] \equiv$
  if $(f = \mu \wedge \mathit{Heap}[\mathsf{Tr}[\![E]\!], \mathit{held}])$
    assert $\mathcal{P}[\mathsf{Tr}[\![E]\!], f] > \varepsilon$;
  else
    assert $\mathcal{P}[\mathsf{Tr}[\![E]\!], f] \geq \varepsilon$;
  $\mathcal{P}[\mathsf{Tr}[\![E]\!], f] := \mathcal{P}[\mathsf{Tr}[\![E]\!], f] - \varepsilon$;

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

$\mathsf{Exhale}[\![P \wedge Q]\!] \equiv$
  $\mathsf{Exhale}[\![Q]\!]$;
  $\mathsf{Exhale}[\![P]\!]$;

$\mathsf{Inhale}[\![P \wedge Q]\!] \equiv$
  $\mathsf{Inhale}[\![P]\!]$;
  $\mathsf{Inhale}[\![Q]\!]$;

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

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

Otherwise:

$\mathsf{Exhale}[\![E]\!] \equiv$
  assert $\mathsf{Tr}[\![E]\!]$;

Otherwise:

$\mathsf{Inhale}[\![E]\!] \equiv$
  assume $\mathsf{Tr}[\![E]\!]$;

Fig. 3. $\mathsf{Exhale}[\![E]\!]$ and $\mathsf{Inhale}[\![E]\!]$ 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.

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:

acquire o; $\equiv$
  assert $\mathit{CanRead}(o.\mathit{shared}) \wedge \mathit{Heap}[o, \mathit{shared}]$;
  assert $\mathit{CanRead}(o.\mu) \wedge (\forall\, p \bullet \mathit{Heap}[p, \mathit{held}] \Rightarrow \mathit{Heap}[p, \mu] \sqsubset \mathit{Heap}[o, \mu])$;
  $\mathit{Heap}[o, \mathit{held}] := \mathbf{true}$;
  $\mathsf{Inhale}[\![J(o)]\!]$

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:

release o; $\equiv$
  assert $o \neq \mathit{null} \wedge \mathit{Heap}[o, \mathit{held}]$;
  $\mathsf{Exhale}[\![J(o)]\!]$
  $\mathit{Heap}[o, \mathit{held}] := \mathbf{false}$;

Finally, the **install** statement requires write permission for $o.\mu$ and that any lower bound $p_i.\mu$ is below any upper bound $s_j.\mu$. It then chooses an appropriate value t 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 p and s, there is some element t strictly in between them:

$$(\forall\, p, s \bullet\ p \sqsubset s\ \Rightarrow\ (\exists\, t \bullet\ p \sqsubset t \wedge t \sqsubset s\ ))$$

Therefore, it is always possible to choose an appropriate value for $o.\mu$. The **foreach** loops can be statically expanded by the translator:

install o **between** $\overline{p}$ **and** $\overline{s}$; $\equiv$
  assert $\mathit{CanWrite}(o, \mu) \wedge \mathit{Heap}[o, \mathit{held}]$;
  foreach $p_i \in \overline{p}, s_j \in \overline{s}$ {
    assert $p_i = \mathit{null} \vee s_j = \mathit{null} \vee$
      $(\mathit{CanRead}(p_i.\mu) \wedge \mathit{CanRead}(s_j.\mu) \wedge \mathit{Heap}[p_i, \mu] \sqsubset \mathit{Heap}[s_j, \mu])$;
  }
  havoc t;
  foreach $p_i \in \overline{p}$ { **assume** $p_i = \mathit{null} \vee \mathit{Heap}[p_i, \mu] \sqsubset t$; } ;
  foreach $s_j \in \overline{s}$ { **assume** $s_j = \mathit{null} \vee t \sqsubset \mathit{Heap}[s_j, \mu]$; } ;
  $\mathit{Heap}[o, \mu] := t$;

Sharing and unsharing. An object o can be shared if the current thread has write permission for $o.\mathit{shared}$ and if the object is not shared already. Like for **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:

share o; $\equiv$
  assert $\mathit{CanWrite}(o, \mathit{shared}) \wedge \neg \mathit{Heap}[o, \mathit{shared}]$;
  Exhale$[\![J(o)]\!]$
  $\mathit{Heap}[o, \mathit{shared}] := \mathbf{true}$;

The **unshare** o statement releases o and at the same time makes it unavailable for further sharing. To prevent further share, it requires full permission for $o.\mu$. Since the **acquire** o statement also requires some permission for $o.\mu$, the two operations are mutually excluded.

unshare o; $\equiv$
  assert $\mathit{CanWrite}(o, \mu) \wedge \mathit{Heap}[o, \mathit{held}]$;
  assert $\mathit{CanWrite}(o, \mathit{shared}) \wedge \mathit{Heap}[o, \mathit{shared}]$;
  $\mathit{Heap}[o, \mathit{shared}] := \mathbf{false}$;
  $\mathit{Heap}[o, \mathit{held}] := \mathbf{false}$;
  havoc $\mathit{Heap}[o, \mu]$;

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 was 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.

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

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:

join o; $\equiv$
  assert $CanWrite(o.active) \wedge Heap[o, active]$;
  $Heap[o, active] := false$;
  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. This is sound because the new thread does not hold any locks and thus, $\mathbf{maxlock} \sqsubset E$ holds trivially.

In App. A.2, we show an example the illustrates reasoning about **fork** and **join**, especially how we use **old**-expressions in the postcondition of the Run method to reason about the effects of a joined thread.

5 Related Work

Implicit dynamic frames were first used by Smans *et al.* [13] as a way to use Kassios's dynamic frames [11] but with access predicates instead of explicit modifies clauses. We extend this work by supporting fractional permissions, which call for the exhale and inhale operations instead of just computing access sets like in [13].

A methodology similar to ours has been defined in separation logic [6]. A difference is that we translate our methodology into first-order verification conditions instead

of needing a separate logic. Checkers for separation logic include Smallfoot, J*, and VeriFast, 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. A minor difference with separation logic is that we can handle **old** expressions, which provide a natural way to write post-conditions. Unlike the work by Gotsman *et al.*, ee also verify that programs do not have deadlocks. On the other hand, we currently have no support for abstract predicates, and we do not check that permissions are not lost.

The technique of specifying the locking order as part of the **share** statement was introduced by Jacobs *et al.*[9]. We extend 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 implemented our methodology in a translator from our experimental source language Chalice to the intermediate verification language Boogie [0] and 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 extend it by two-state invariants to permit rely-guarantee reasoning and by abstraction via user-defined functions or predicates. We also want to extend Chalice to a full object-oriented language by adding subtyping, which we expect to be straightforward. Another important feature to add to Chalice is abstract predicates, which we plan to investigate in for implicit dynamic frames with fractional permissions.

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 F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, Sept. 2006.

1. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *Principles of Programming Languages (POPL)*, volume 40(1), pages 259–270. ACM, 2005.
2. J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis (SAS)*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.
3. P. Brinch Hansen. *Operating systems principles*. Prentice-Hall, Englewood Cliffs, NJ, 1973.
4. L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
5. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, May 2005.
6. A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *APLAS'07: Asian Symposium on Programming Languages and Systems*, volume 4807 of *Lecture Notes in Computer Science*, pages 19–37. Springer, 2007.
7. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
8. C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, Oct. 1974.
9. B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A statically verifiable programming model for concurrent object-oriented programs. In Z. Liu and J. He, editors, *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006*, volume 4260 of *Lecture Notes in Computer Science*, pages 420–439. Springer, Nov. 2006.
10. C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332. North-Holland, 1983.
11. I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, Aug. 2006.
12. D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 1999.
13. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. In *Formal Techniques for Java-like Programs (FTfJP 2008)*, 2008. To appear.
14. D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM Trans. Prog. Lang. Syst.*, 22(4):701–771, 2000.
15. 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.

A Examples

In this appendix, we present additional examples that highlight interesting aspects of our methodology.

A.0 Hand-over-hand locking

Method *Insert* of class *List* (Fig. 2) inserts a new value into the list. It uses 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 we traverse the list. Hand-over-hand locking becomes possible by our use of fractional permissions in the monitor invariant of *Node*.

```
void Insert(int x)
  requires rd(μ) ∧ maxlock ⊏ μ ∧ rd(shared) ∧ shared ∧ acc(sum, 80) ∧ 0 ≤ x ;
  ensures rd(shared) ∧ shared ∧ acc(sum, 80) ∧ sum = old(sum) + x ;
{
  acquire this ;
  sum := sum + x ;
  Node p := head ;
  acquire p ;
  p.sum := p.sum + x ;
  release this ;
  while (p.next ≠ null ∧ p.next.val < x)
    invariant p ≠ null ∧ acc(p.next) ∧ rd(p.val) ∧ acc(p.μ, 50) ;
    invariant p.held ∧ maxlock = p.μ ;
    invariant p.next ≠ null ⇒ acc(p.next.shared) ∧ p.next.shared ;
    invariant p.next ≠ null ⇒ acc(p.next.μ, 50) ∧ p.μ ⊏ p.next.μ ;
    invariant p.next ≠ null ⇒ rd(p.next.val) ∧ p.val ≤ p.next.val ;
    invariant acc(p.sum, 50) ∧ (p.next = null ⇒ p.sum = x) ;
    invariant p.next ≠ null ⇒ acc(p.next.sum, 50) ∧
                               p.sum = p.next.val + p.next.sum + x ;
    invariant p.val ≤ 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 ;
}
```

A.1 Dynamic change of locking order

The following example illustrates dynamic changes of the locking order. $AssocList$ implements a linked list that stores key-value pairs. Each time an entry is looked up in method Get, it is moved toward the front of the list, hoping that this popular entry will be found faster next time. Since the locking order corresponds to the order the nodes occur in the list (see $SNode$'s second monitor invariant), a change of the list structure also requires a change of the locking order. Method Get uses the **install** statement to re-arrange the locking order. According to the loop invariant, the current thread has full permission for $p.next.\mu$ and read permission for $p.next.next.\mu$; moreover, $p.next$ is ordered below $p.next.next$. Therefore, the requirements for **install** are satisfied. Note that in this example, one could also move the entry by exchanging the keys and values rather than by swapping the nodes. However, other example such as list reversal are typically implemented by changing the node structure.

```
class Data { }

class SNode {
  int key ;
  Data value ;
  SNode next ;

  invariant rd(key) ∧ rd(value) ∧ acc(next) ∧ acc(μ, 50) ;
  invariant next ≠ null ⇒ acc(next.μ, 50) ∧ μ ⊏ next.μ ∧
                          rd(next.shared) ∧ next.shared ;
}

class AssocList {
  SNode head ;    // sentinel node

  invariant rd(head) ∧ head ≠ null ;
  invariant rd(head.μ) ∧ μ ⊏ head.μ ∧ rd(head.shared) ∧ head.shared ;

  void Init()
    requires acc(head) ∧ acc(μ) ∧ acc(shared) ∧ ¬shared ;
    ensures acc(μ) ∧ acc(shared) ∧ shared ;
  {
    head := new SNode ;
    head.next := null ;
    share head ;
    install head between this and  ;
    share this ;
  }
```

```
  Data Get(int key)
    requires rd(μ) ∧ maxlock ⊏ μ ∧ rd(shared) ∧ shared ;
    ensures rd(μ) ∧ rd(shared) ;
  {
    Data d := null ;
    acquire this ;
    SNode p := head ;
    acquire p ;
    release this ;

    if (p.next ≠ null) {
      acquire p.next ;
      if (p.next.key = key) {
      d := p.next.value ;
    } else {
      bool done := false ;
      while (¬done)
        invariant p ≠ null ∧ rd(p.key) ∧ rd(p.value) ∧ acc(p.next) ∧ acc(p.μ, 50) ;
        invariant p.next ≠ null ∧ acc(p.next.μ) ∧ p.μ ⊏ p.next.μ ;
        invariant rd(p.next.shared) ∧ p.next.shared ;
        invariant rd(p.next.key) ∧ rd(p.next.value) ∧ acc(p.next.next) ;
        invariant p.next.next ≠ null ⇒
                                  acc(p.next.next.μ, 50) ∧ p.next.μ ⊏ p.next.next.μ ∧
                                  rd(p.next.next.shared) ∧ p.next.next.shared ;
        invariant p.held ∧ p.next.held ∧ maxlock = p.next.μ ;
      {
        if (p.next.next = null) {
          done := true ;    // key not present
        } else {
          acquire p.next.next ;
          if (p.next.next.key = key) {
            done := true ;    // key is present
            d := p.next.next.value ;
            // move p.next.next closer to the head by one step
            SNode t := p.next ;
            p.next := t.next ; t.next := p.next.next ; p.next.next := t ;
            install t between p.next and t.next ;
            release t ;
          } else {
            SNode t := p ;
            p := p.next ;
            release t ;
          }
        }
      }
    }
    release p.next ; release p ;
    return d ;
  }
}
```

A.2 Fork-join

The following example illustrates that our methodology enables one to prove strong properties about forked and joined threads. Method $ForkJoin$ forks a new thread on object x, which, according to the precondition of Run, obtains full permission for $x.k$, but no permission for $x.l$. Therefore, the forking thread is allowed to update $x.l$ after the fork. After joining the previously forked thread, we can conclude from Run's postcondition that $x.k = 18$. Note that we can draw this conclusion even though the Inhale operation of the **fork** havocs that location, because the **old**-expression in the postcondition of Runr refers to the state before the **havoc**, for which we know $x.k = 17$. Also notice that it is possible in $JustJoin$ to figure out that $x.k$ is at least 1 after the join, despite that fact that the matching fork is not in sight. It is sound to conclude $1 \leq x.k$ after the join, since the thread precondition says that $x.k$ had to start off being at least 0.

```
class T {
  int k ;
  int l ;

  void Run()
    requires acc(k) ∧ 0 ≤ k ;
    ensures acc(k) ∧ k = old(k) + 1 ;
  {
    k := k + 1 ;
  }

  int ForkJoin()
    ensures result = 8 ;
  {
    T x := new T ;
    x.k := 17 ;
    x.l := 20 ;
    fork x ;
    x.l := 10 ;
    join x ;
    assert x.k = 18 ∧ x.l = 10 ;
    return x.k − x.l ;
  }

  int JustJoin(T x)
    requires x ≠ null ∧ acc(x.active) ∧ x.active ;
    ensures 1 ≤ result ;
  {
    join x ;
    return x.k ;
  }
}
```