The Repeat Offender Problem: A Mechanism for Supporting Dynamic-sized Lock-free Data Structures

Maurice Herlihy, Victor Luchangco, and Mark Moir

The Repeat Offender Problem: A Mechanism for Supporting Dynamic-sized Lock-free Data Structures

Maurice Herlihy, Victor Luchangco, and Mark Moir

SMLI TR-2002-112 July 2002

Abstract:

We define the *Repeat Offender Problem* (ROP). Elsewhere, we have presented the first dynamic-sized lock-free data structures that can free memory to any standard memory allocator—even after thread failures—without requiring special support from the operating system, the memory allocator, or the hardware. These results depend on a solution to the ROP problem. Here we present the first solution to the ROP problem and its correctness proof. Our solution is implementable in most modern shared memory multiprocessors.

M/S MTV29-01
2600 Casey Avenue
Mountain View, CA 94043

email addresses:
mph@cs.brown.edu
victor.luchangco@sun.com
mark.moir@sun.com

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@sun.com>. All technical reports are available online on our Website, http://research.sun.com/techrep/.

The Repeat Offender Problem: A Mechanism for Supporting Dynamic-sized Lock-free Data Structures

Maurice Herlihy
Computer Science Department
Brown University, Box 1910
Providence, RI 02912

Victor Luchangco Mark Moir
Sun Microsystems Laboratories
1 Network Drive, UBUR02-311
Burlington, MA 01803

1 Introduction

This paper presents a mechanism for supporting memory management in dynamic-sized lock-free data structures (i.e., those that can grow and shrink). Lock-free data structures avoid many problems associated with the use of locking, including convoying, susceptibility to failures and delays, and, in real-time systems, priority inversion. A *lock-free* data structure guarantees that after a finite number of steps of any operation on the data structure, *some* operation completes. This definition precludes the use of locks to protect the data structure because a thread can take an unbounded number of steps without completing an operation if some other thread is delayed or fails while holding a lock that the first thread requires. The difficulty of designing lock-free data structures is reflected in numerous papers in the literature describing clever and subtle algorithms for implementing relatively mundane data structures such as stacks [13], queues [10], and linked lists [14, 5].

We define an abstract problem—the *Repeat Offender Problem* (ROP)—such that any solution to this problem can be used by dynamic-sized lock-free data structure implementations to allow them to free unused memory with standard memory allocators. We have paid particular attention to formulating our problem to support one or more "worker" threads that do most of the work of memory management. This allows us to separate the scheduling of this work from that of the application. In particular, it allows us to use "spare" processors to do memory management work.

In this paper, we present the first solution to the ROP problem, which we call "Pass The Buck". Both the problem statement and our solution are quite involved, and the algorithm in particular is subtle and requires careful explanation (and a formal proof, which is included in an appendix). In a separate report [6], we show how to apply ROP solutions to achieve the first[1] truly dynamic-sized lock-free data structures, and evaluate their performance. In the remainder of this section, we discuss why dynamic-sized data structures are challenging to implement in a lock-free manner and then briefly summarize previous related work.

Before freeing an object that is part of a data structure (e.g., a node of a linked list), we must ensure that no thread will subsequently modify the object. Otherwise, a thread might corrupt an object allocated later that happens to reuse the memory used by the first object. Furthermore, in some systems, even read-only accesses of freed objects can be problematic: the operating system

[1] Concurrently and independently, Maged Michael has developed a technique that is similar to ours [9]. We discuss differences between the two approaches at the end of Section 3.

may remove the page containing the object from the thread's address space, causing a subsequent access to crash the program because the address is no longer valid [13].

The use of locks makes it relatively easy to ensure that freed objects are not subsequently accessed because we can prevent access by other threads to (parts of) the data structure while removing objects from it. In contrast, without locks, multiple operations may access the data structure concurrently, and a thread cannot determine whether other threads are already committed to accessing the object that it wishes to free (this can only be ascertained by inspecting the stacks and registers of other threads). This is the root of the problem that our work aims to address.

Below we discuss various previous approaches for dealing with the problem described above.[2] One easy approach is to use garbage collection (GC). GC ensures that an object is not freed while any pointer to it exists, so threads cannot access objects after they are freed. This approach is especially attractive because recent experience (e.g., [2]) shows that GC significantly simplifies the design of dynamic-sized lock-free data structures. However, GC is not available in all languages and environments, and in particular, we cannot rely on GC to implement GC.

Another common approach is to augment values in objects with version numbers or tags, and to access such values only through the use of compare-and-swap (CAS), such that if a CAS executes on an object after it has been deallocated, the value of the version number or tag will ensure that the CAS fails [13, 11, 12]. In this case, the version number or tag value must be carried with the object through deallocation and reallocation. A common approach for achieving this is to maintain explicit "freelists", which contain objects that are not currently in use [13, 11]. (In the remainder of the paper, we refer to freelists as "object pools" in order to avoid confusing "freeing" an object—by which we mean returning it to the memory allocator through the **free** library routine—with placing an object on a freelist.) In this approach, rather than freeing objects to the memory allocator when they are no longer required, we place them in an object pool from which new objects of the same type can be allocated later. An important disadvantage of this approach is that data structures implemented this way are not truly dynamic-sized: after they have grown large and subsequently shrunk, the object pool contains many objects that cannot be reused for other purposes, cannot be coalesced, etc. In [6], we show how to eliminate the need for object pools from Michael and Scott's lock-free FIFO queue implementation using the results presented here. There are also various performance-related *advantages* to using object pools, rather than always allocating from and freeing to the memory allocator. We therefore also show in [6] how to construct object pools whose elements can be freed to the memory allocator. Thus, we show how to eliminate the major disadvantage of previous object pool approaches without sacrificing their advantages. We also present performance results in [6] that show that the overhead of making these data structures dynamic-sized is negligible in the absence of contention, and low in all cases. We believe these are the first dynamic-sized lock-free data structures that can continue to reclaim memory even after threads fail.

Valois [14] proposed another approach, in which the memory allocator maintains reference counts for objects to determine when they can be freed. Valois's approach allows the reference count of an object to be accessed even after the object has been released to the memory allocator. This behaviour restricts what the memory allocator can do with released objects. For example, the released objects cannot be coalesced. Thus, the disadvantages of maintaining explicit object pools are shared by Valois's approach. Furthermore, application designers sometimes need to switch between different memory allocation implementations for performance or other reasons. Valois's

[2] These approaches are all forms of *type stable memory* (TSM), defined by Greenwald [4] as follows: "TSM [provides] a guarantee that an object O of type T remains type T as long as a pointer to O exists."

approach requires the memory allocator to support certain nonstandard functionality, and therefore restricts this possibility. Finally, the space overhead for per-object reference counts may be prohibitive. (In [3], we proposed a similar approach that does allow memory allocators to be interchanged, but depends on double compare-and-swap (DCAS), which is not widely supported.)

Our goal is to provide support for the design of dynamic-sized lock-free data structures that can free objects to the memory allocator through standard interfaces (so memory allocators can be switched with ease), and can ensure that freed objects are not subsequently accessed (so it is safe for the memory allocator to unmap pages containing previously freed objects).

Interestingly, the previous work that comes closest to meeting this goal predates the work discussed above by almost a decade. Treiber [13] proposes a technique called *obligation passing*.[3] The instance of this technique for which Treiber presents specific details is in the implementation of a lock-free linked list supporting search, insert, and delete operations. This implementation allows freed nodes to be returned to the memory allocator through standard interfaces and without requiring any special functionality of the memory allocator. However, it employs a "use counter" such that memory is reclaimed only by the "last" thread to access the linked list in any period. As a result, this implementation can be prevented from ever recovering any memory by a failed thread (which defeats one of the main purposes of using lock-free implementations). Another disadvantage of this implementation is that the obligation passing code is bundled together with the linked-list maintenance code (all of which is presented in assembly code). Because it is not clear what aspects of the linked-list code the obligation passing code depends on, it is difficult to apply this technique to other situations.

The remainder of this paper is structured as follows. In Section 2, we define ROP, and discuss some of the issues and trade-offs we faced in specifying the problem. In Section 3, we present the Pass The Buck algorithm. Concluding remarks appear in Section 4. A formal correctness proof for the Pass The Buck algorithm is presented in an appendix.

2 The Repeat Offender Problem

In this section, we specify the *Repeat Offender Problem* (ROP). We consider a set of *values*, each of which can be free, injail, or escaping. Initially, all values are free. We further consider a set of *application clients* that interact with an ROP solution as described below. An application-dependent external Arrest action can cause a free value to become injail at any time. A client can help injail values to begin escaping, which causes them to become escaping. Values that are escaping can finish escaping and become free again. Clients can use values, but must never use a value that is free. A client can attempt to prevent a value v from escaping while it is being used by "posting a guard" on v. However, if the guard is posted too late, it may fail to prevent v from escaping. Thus, in order to safely use v, a client first posts a guard on v, and then checks to see if v is still injail.[4] If so, then an ROP solution is required to ensure that v does not escape before the guard is stood down or posted elsewhere.

Our motivation is to use ROP solutions to allow threads (clients) to avoid dereferencing (using)

[3] We named our technique "Pass The Buck" before we were aware of Treiber's work on obligation passing [13], which indicates the similarity of the underlying philosophies of our approaches, although the details are quite different.

[4] In some cases, it is possible for a client p to determine independently of ROP that a value it wants to use will remain injail until p uses the value. In this case, p can use the value without posting a guard on it. An example of such an optimization is presented in [6].

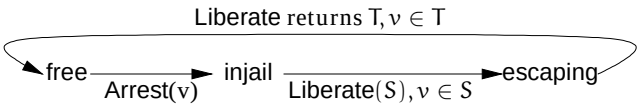

Figure 1: Transition diagram for value v.

a pointer (value) to an object that has been freed. In this context, an injail pointer is one that has been allocated (arrested) since it was last freed, and can therefore be used.

To support these interactions, ROP solutions provide the following procedures. A thread *posts* a guard g on a value v by invoking PostGuard(g,v); this removes the guard from any value it previously guarded. (A special **null** value is used to *stand down* the guard, that is, to remove the guard from the previously guarded value without posting the guard on a new value). A thread helps a set of values S to *begin escaping* by invoking Liberate(S); the application must ensure that each value in S is injail before this call, and the call causes each value to become escaping. The Liberate procedure returns a (possibly different) set of escaping values causing them to be *liberated* (each of these values becomes free on the return of this procedure). These transitions are summarized in Figure 1. Finally, a thread can check whether a value v is injail by invoking IsInJail(v); if this invocation returns *true*, then v was injail at some point during the invocation[5] (the converse is not necessarily true, as explained later). An ROP solution does not implement the functionality of the Arrest action—this is application-specific—but ROP must be aware of arrests to know when a free value becomes injail.

If a guard g is posted on a value v, and v is injail at some time t after g is posted on v and before g is subsequently stood down or reposted on a different value, then we say that g *traps* v from time t until g is stood down or reposted. The operational specification of the main correctness condition for ROP is that it does not allow a value to escape (i.e., become free) while it is trapped.

The above description of ROP omitted some important but mundane details. We now discuss these details and then present a formal specification of ROP that includes them. First, we did not describe how clients choose guards to post on values. In some applications (an example is presented in [6]), it is necessary for one client to guard multiple values at the same time, so simply having one guard per client is not sufficient. We therefore allow clients to *hire* and *fire* guards. Guards are hired by invoking the HireGuard procedure and are fired by invoking the FireGuard procedure. A client may not post, stand down or fire a guard that it does not currently employ. An ROP solution is required to ensure that a guard is never simultaneously employed by multiple clients.

The above description should be sufficient for a basic understanding of ROP; a precise formulation of ROP is given by the I/O automaton shown in Figure 2, explained below. (See [8] for details of the I/O automata model.) We begin by adopting some notational conventions.

Notational conventions: Unless otherwise specified, p and q denote clients (threads) from P, the set of all clients (threads); g denotes a guard from G, the set of all guards; v denotes a value from V, the set of all values, and S and T denote sets of values (i.e., subsets of V). We assume that V

[5]Our formal specification requires this point to coincide with the response of the IsInJail(v) operation. Technically, we should have a separate internal action that observes the status of v. However, because in practice clients cannot determine the point at which the response action occurred, and therefore cannot distinguish between implementations that meet these two specifications, we did not bother with this technicality.

actions

Environment	ROP output
$\mathsf{HireInv}_p()$	$\mathsf{HireResp}_p(g)$
$\mathsf{FireInv}_p(g)$	$\mathsf{FireResp}_p()$
$\mathsf{PostInv}_p(g,v)$	$\mathsf{PostResp}_p()$
$\mathsf{IsInJailInv}_p(v)$	$\mathsf{IsInJailResp}_p(b)$
$\mathsf{LiberateInv}_p(S)$	$\mathsf{LiberateResp}_p(S)$
$\mathsf{Arrest}(v)$	

transitions

$\mathsf{HireInv}_p()$
Pre: $pc_p = \mathsf{idle}$
Eff: $pc_p \leftarrow \mathsf{hire}$

$\mathsf{FireInv}_p(g)$
Pre: $pc_p = \mathsf{idle}$
$g \in guards_p$
$post[g] = \textbf{null}$
Eff: $pc_p \leftarrow \mathsf{fire}$
$guards_p \leftarrow guards_p - \{g\}$

$\mathsf{PostInv}_p(g,v)$
Pre: $pc_p = \mathsf{idle}$
$g \in guards_p$
Eff: $pc_p \leftarrow \mathsf{post}(g,v)$
$post[g] \leftarrow \textbf{null}$
$trapping[g] \leftarrow \mathrm{false}$

$\mathsf{IsInJailInv}_p(v)$
Pre: $pc_p = \mathsf{idle}$
Eff: $pc_p \leftarrow \mathsf{injail}(v)$

$\mathsf{LiberateInv}_p(S)$
Pre: $pc_p = \mathsf{idle}$
for all $v \in S$,
$v \neq \textbf{null}$ and $status[v] = \mathsf{injail}$
Eff: $pc_p \leftarrow \mathsf{liberate}$
$numescaping \leftarrow numescaping + |S|$
for all $v \in S$, $status[v] \leftarrow \mathsf{escaping}$

$\mathsf{Arrest}(v)$
Pre: $status[v] = \mathsf{free}$
$v \neq \textbf{null}$
Eff: $status[v] \leftarrow \mathsf{injail}$
for all g such that $post[g] = v$,
$trapping[g] \leftarrow \mathrm{true}$

state variables
For each client $p \in P$:
pc_p: {idle, hire, fire, post(g,v), injail(v), liberate} **init** idle
$guards_p$: set of guards **init** empty
For each value $v \in V$:
$status[v]$: {injail, escaping, free} **init** free
For each guard $g \in G$:
$post[g]$: V **init null**;
$trapping[g]$: **bool init** false;
$numescaping$: **int init** 0

$\mathsf{HireResp}_p(g)$
Pre: $pc_p = \mathsf{hire}$
$g \in G$
$g \notin \bigcup_q guards_q$
Eff: $pc_p \leftarrow \mathsf{idle}$
$guards_p \leftarrow guards_p \cup \{g\}$

$\mathsf{FireResp}_p()$
Pre: $pc_p = \mathsf{fire}$
Eff: $pc_p \leftarrow \mathsf{idle}$

$\mathsf{PostResp}_p()$
Pre: for some g, v, $pc_p = \mathsf{post}(g,v)$
Eff: $pc_p \leftarrow \mathsf{idle}$
$post[g] \leftarrow v$
$trapping[g] \leftarrow (status[v] = \mathsf{injail})$

$\mathsf{IsInJailResp}_p(b)$
Pre: for some v, $pc_p = \mathsf{injail}(v)$
$b \Rightarrow (status[v] = \mathsf{injail})$
Eff: $pc_p \leftarrow \mathsf{idle}$

$\mathsf{LiberateResp}_p(S)$
Pre: $pc_p = \mathsf{liberate}$
for all $v \in S$,
$status[v] = \mathsf{escaping}$
and for all $g \in \bigcup_q guards_q$,
($post[g] \neq v$ or $\neg trapping[g]$)
Eff: $pc_p \leftarrow \mathsf{idle}$
$numescaping \leftarrow numescaping - |S|$
for all $v \in S$, $status[v] \leftarrow \mathsf{free}$

Figure 2: I/O Automaton specifying the Repeat Offender Problem.

contains a special **null** value that is never used, arrested or liberated. □

The automaton consists of a set of environment actions and a set of ROP output actions. Each action consists of a *precondition* for performing the action and the *effect* on state variables of performing the action. Most environment actions are invocations of ROP operations and are paired with corresponding ROP output actions that represent the system's response to the invocations. In particular, the $\mathsf{PostInv}_p(g, v)$ action models client p invoking PostGuard(g,v), and the $\mathsf{PostResp}_p()$ action models the completion of this procedure. The $\mathsf{HireInv}_p()$ action models client p invoking HireGuard(), and the corresponding $\mathsf{HireResp}_p(g)$ action models the system assigning guard g to p. The $\mathsf{FireInv}_p(g)$ action models client p calling FireGuard(g), and the $\mathsf{FireResp}_p()$ action models the completion of this procedure. The $\mathsf{LiberateInv}_p(S)$ action models client p calling Liberate(S) to help the values in S begin escaping, and the $\mathsf{LiberateResp}_p(T)$ action models the completion of this procedure with a set of values T that have finished escaping. Finally, the Arrest(v) action models the environment arresting value v.

The state variable *status*[v] records the current status of value v, which can be free, injail, or escaping. Transitions between the status values are caused by calls to and returns from ROP procedures, as well as by the application-specific Arrest action, as described above. The *post* variable maps each guard to the value (if any) it currently guards. The pc_p variable models control flow of client p, for example ensuring that p does not invoke a procedure before the previous invocation completes; pc_p also encodes parameters passed to the corresponding procedures in some cases. The $guards_p$ variable represents the set of guards currently employed by client p. The *numescaping* variable is an auxiliary variable used to specify nontriviality properties, as discussed later. Finally, *trapping* maps each guard g to a boolean value that is *true* iff g has been posted on some value v and has not subsequently been reposted or stood down, and at some point since the guard was posted on v, v has been injail (i.e., it captures the notion of guard g trapping the value on which it has been posted). This is used by the LiberateResp action to determine whether v can be returned. (Recall that a value should not be returned if it is trapped.)

Preconditions on the invocation actions specify assumptions about the circumstances under which the application invokes the corresponding ROP procedures. Most of these preconditions are mundane well-formedness conditions such as the requirement that a client posts only guards that it currently employs. The precondition for LiberateInv captures the assumption that the application passes only injail values to Liberate, and the precondition for the Arrest action captures the assumption that only free values are arrested. The application designer must determine how these guarantees are made.

Preconditions on the response actions specify the circumstances under which the ROP procedures can return. Again, most of these preconditions are quite mundane and straightforward. The interesting case is the precondition of LiberateResp, which states that Liberate can return a value only if it has been passed to (some invocation of) Liberate, it has not subsequently been returned by (any invocation of) Liberate, and no guard g has been continually guarding the value since it was last injail (recall that this is captured by *trapping*[g]).

Desirable properties

As specified so far, an ROP solution in which Liberate always returns the empty set, or simply does not terminate, is correct. Clearly, in contexts such as that motivating our work, such solutions are unacceptable: each escaping value represents a resource that will be reclaimed only when the

value is liberated (returned by some invocation of Liberate). One might be tempted to specify that every value passed to a Liberate operation is eventually returned by some Liberate operation. However, without special operating system support, it is not possible to guarantee such a strong property in the face of failing threads. We do not attempt here to specify the "correct" nontriviality condition, as we do not want to unduly limit the range of solutions. Instead, we discuss some properties that might be useful in specifying nontriviality properties for proposed solutions.

The state variable *numescaping* counts the number of values that are currently escaping (i.e., that have been passed to some invocation of Liberate and have not subsequently been returned from any invocation of Liberate). If we require a solution to ensure that *numescaping* is bounded by some function of application-specific quantities, we exclude the trivial solution in which Liberate always returns the empty set. However, because this bound necessarily depends on the number of concurrent Liberate operations and the number of values with which each Liberate operation is invoked, it does not exclude the solution in which Liberate never returns.

A combination of a boundedness requirement and some form of progress requirement on Liberate operations seems to be the most appropriate way to specify the nontriviality requirement. We later prove that the Pass The Buck algorithm provides a bound on *numescaping* that depends on the number of concurrent Liberate operations. Because the bound (necessarily) depends on the number of concurrent Liberate operations, if an unbounded number of threads fail while executing Liberate, then an unbounded number of values can be escaping. We emphasize, however, that our implementation does *not* allow failed threads to prevent values from being freed in the future, as Treiber's approach does [13].

Our Pass The Buck algorithm has two more desirable properties. First, the Liberate operation is wait-free (that is, it completes after a bounded number of steps, regardless of the timing behaviour of other threads). This is useful because it allows us to calculate an upper bound on the amount of time Liberate will take to execute, which is useful in determining how to schedule Liberate work.

Finally, our algorithm has a property we call *value progress*. Roughly, this property guarantees that, unless a thread fails, a value does not remain escaping forever provided Liberate is invoked "enough" times.

Modular decomposition

A key contribution of this paper is the insight that an effective way to solve ROP in practice is to separate the implementation of the IsInJail operation from the others. The reason is that, in our experience using ROP solutions to implement dynamic-sized lock-free data structures [6], values are used in a manner that allows threads to efficiently determine whether a value is injail *with sufficient accuracy for the particular application*. As a concrete example, when values represent pointers to objects that are part of a concurrent data structure, these values become injail (allocated) before the objects they refer to become part of the data structure, and are removed from the data structure before being passed to Liberate. Thus, simply observing that an object is still part of a data structure is sufficient to conclude that a pointer to it is injail.

Because we intend ROP solutions to be used with application-specific implementations of the IsInJail operation, the specification of this operation is somewhat weak: an implementation of IsInJail that always returns *false* meets the specification. However, this implementation would be useless, usually because it would not guarantee the required progress properties of the application that uses it. Because the circumstances under which IsInJail can and should return *true*¡ depend on

the application, we retain the weak specification of IsInJail, and leave it to application designers to provide implementations of IsInJail that are sufficiently strong for their applications. (Note that an integrated, application-independent implementation of this operation, while possible, would be expensive: it would have to monitor and synchronize with all actions that potentially affect the status of each value.)

This proposed modular decomposition suggests the following methodology for implementing dynamic-sized lock-free objects: Use an "off-the-shelf" implementation of an ROP solution for the HireGuard, FireGuard, PostGuard, and Liberate operations, and then exploit specific knowledge of the application to design an optimized implementation of IsInJail. More precisely, we decompose the ROP I/O automaton into two component automata: the *ROPlite* automaton, and the *InJail* automaton. *ROPlite* has the same environment and output actions as ROP, except for IsInJailInv and IsInJailResp. *InJail* has input action IsInJailInv and output action IsInJailResp. In addition, the *InJail* automaton "eavesdrops" on *ROPlite*: all environment and output actions of *ROPlite* are input actions of *InJail* (though in many cases the implementation of the *InJail* automata will ignore these inputs because it can determine whether a value is injail without them, as discussed above).

We present our Pass The Buck algorithm, which implements *ROPlite* in a simple and practical way, in Section 3.

Issues and tradeoffs in specifying ROP

We could have rolled the functionality of hiring and firing guards into the PostGuard operation. We kept this functionality separate in order to allow implementations to make the PostGuard operation as efficient as possible, as this is the most common operation. This separation allows the implementation more flexibility in how it manages resources associated with guards because the cost of hiring and firing guards can be amortized over many PostGuard operations.

In some applications, it may be desirable to be able to quickly "mark" a value for liberation, without doing any of the work of liberating the value. (Consider, for example, an interactive system in which user threads should not execute relatively high-overhead "administrative" work such as liberating values, but additional processor(s) may be available to perform such work.) We did not model such an operation, as it is straightforward to communicate such values to a worker thread that invokes Liberate—the ROP solution does not need to know anything about this.

3 One Solution: Pass The Buck

In this section, we describe one ROP solution. Our primary goal when designing this solution was to minimize the performance penalty to the application when no values are being liberated. That is, the PostGuard operation should be implemented as efficiently as possible, perhaps at the cost of a more expensive Liberate operation. Such solutions are desirable for at least two reasons. First, PostGuard is necessarily invoked by the application, so its performance always impacts application performance. On the other hand, Liberate work can be done by a spare processor, or by a background thread, so that it does not directly impact application performance. Second, solutions that optimize PostGuard performance are desirable for scenarios in which values are liberated infrequently, but we must retain the ability to liberate them. An example is the implementation of a dynamic-sized data structure that uses an object pool to avoid allocating and freeing objects under "normal" circumstances but can free elements of the object pool when it grows too large.

```
struct { value val; int ver } HO_t
    // HO_t fits into CAS-able location
constant MG: max. number of guards
shared variable
  GUARDS: array[0..MG-1] of bool init false;
  MAXG: int init 0;
  POST: array[0..MG-1] of value init null;
  HNDOFF: array[0..MG-1] of HO_t init ⟨null, 0⟩;

int HireGuard() {
1   int i = 0, max;
2   while (!CAS(&GUARDS[i],false,true))
3     i++;
4   while ((max = MAXG) < i)
5     CAS(&MAXG,max,i);
6   return i;
}

void FireGuard(int i) {
7   GUARDS[i] = false;
8   return
}

void PostGuard(int i, value v) {
9   POST[i] = v;
10  return
}

value_set Liberate(value_set vs) {
11  int i = 0;
12  while (i <= MAXG) {
13    int attempts = 0;
14    HO_t h = HNDOFF[i];
15    value v = POST[i];
16    if (v != null && vs→search(v)) {
17      while (true) {
18        if (CAS(&HNDOFF[i], h, ⟨v, h.ver+1⟩)) {
19          vs→delete(v);
20          if (h.val != null) vs→insert(h.val);
21          break;
          }
22        attempts++;
23        if (attempts == 3) break;
24        h = HNDOFF[i];
25        if (attempts == 2 && h.val != null) break;
26        if (v != POST[i]) break;
        }
27    } else {
28      if (h.val != null && h.val != v)
29        if (CAS(&HNDOFF[i], h, ⟨null, h.ver+1⟩))
30          vs→insert(h.val);
      }
31    i++;
    }
32  return vs;
}
```

Figure 3: Code for Pass The Buck.

In this case, no liberating is necessary while the size of the data structure is relatively stable. With this goal in mind, we describe our Pass The Buck algorithm below, after some preliminaries.

Preliminaries: Our algorithm is presented in a C/C++-like pseudocode style, and should be self-explanatory. For convenience, we assume a shared-memory multiprocessor with sequentially consistent memory [7].[6] We further assume that the multiprocessor supports a compare-and-swap (CAS) instruction that accepts three parameters: an *address*, an *expected* value, and a *new* value. The CAS instruction atomically compares the contents of the address to the expected value, and, if they are equal, stores the new value at the address and returns *true*. If the comparison fails, no changes are made to memory, and the CAS instruction returns *false*. □

The Pass The Buck algorithm is shown in Figure 3. The GUARDS array is used to allocate guards to threads. Here we assume a bound MG on the number of guards simultaneously employed; it is straightforward to remove this restriction. The POST array consists of one location per guard, which holds the value the guard is currently assigned to guard if one exists, and **null** otherwise. The HNDOFF array is used by Liberate to "hand off" responsibility for a value to a later Liberate operation if the value cannot be liberated immediately because it has been trapped by a guard.

The HireGuard and FireGuard procedures essentially implement long-lived renaming; we use

[6] We have implemented our algorithms for SPARC®-based machines providing only TSO (Total Store Ordering) [15]—a memory model that is slightly weaker than sequential consistency—which required additional memory barrier instructions to be included in places.

the renaming algorithm presented in [1]. Specifically, for each guard g, we maintain an entry GUARDS[g], which is initially *false*. Thread p hires guard g by atomically changing GUARDS[g] from *false* (unemployed) to *true* (employed); p attempts this with each guard in turn until it succeeds (lines 2 and 3). The FireGuard procedure simply sets the guard back to *false* (line 7). The HireGuard procedure also maintains the shared variable MAXG, which is used by the Liberate procedure to determine how many guards to consider. Liberate considers every guard for which a HireGuard operation has completed. Therefore, it suffices to have each HireGuard operation ensure that MAXG is at least the index of the guard returned. This is achieved with the simple loop at lines 4 and 5.

PostGuard is implemented as a single store of the value to be guarded in the specified guard's POST entry (line 9), in accordance with our goal of making PostGuard as efficient as possible.

The most interesting part of the Pass The Buck algorithm lies in the Liberate procedure. Recall that Liberate should return a set of values that have been passed to Liberate and have not since been returned by Liberate (i.e., escaping values), subject to the constraint that Liberate cannot return a value that has been continuously guarded by the same guard since some point when it was injail (i.e., Liberate must not return trapped values).

Because we want the Liberate operation to be wait-free, if some guard g is guarding a value v in the value set of some thread p executing Liberate, then p must either determine that g is not trapping v or remove v from p's value set before returning that set. To avoid losing values, any value that p removes from its set must be stored somewhere so that, when the value is no longer trapped, another Liberate operation may pick it up and return it. The interesting details of the Pass The Buck algorithm concern how threads determine that a value is not trapped, and how they store values while keeping space overhead for stored values low. Below we explain the Liberate procedure in more detail, paying particular attention to these issues.

The loop at lines 12 through 31 iterates over all guards ever hired. For each guard, if p cannot determine for some value v in its set that v is not trapped, then p attempts to "hand off" that value (there can be at most one such value per guard). If p succeeds in doing so (line 18), it removes v from its set (line 19) and proceeds to the next guard (lines 21 and 31). Also, as explained in more detail below, p might simultaneously pick up a value handed off previously by another Liberate operation, in which case this value can be shown not to be trapped by that guard, so p adds this value to its set (line 20). If p fails to hand v off, then it retries. If it fails repeatedly, it can be shown that v is not trapped by that guard, so p can move on to the next guard without removing v from its set (lines 23 and 25). When p has examined all guards (see line 12), it can safely return any values remaining in its set (line 32).

We describe the processing of each guard in more detail below. First, however, we present a central property of the correctness proof of this algorithm, which will aid the presentation that follows; this lemma is quite easy to see from the code and the high-level description given thus far; it is formalized in Invariant 12 of the correctness proof given in the appendix.

Single Location Lemma: Each escaping value v is stored at a single guard or is in the value set of a single Liberate operation (but not both). Also, only escaping values are in any of these locations.

The processing of each guard g proceeds as follows: At lines 15 and 16, p determines whether the value currently guarded by g (if any)—call it v—is in its set. If so, p executes the loop at lines 17 through 26 in order to either determine that v is not trapped, or to remove v from its set. In order to avoid losing v in the latter case, p "hands off" v by storing it in the HNDOFF array.

(Each entry of this array consists of a value and a version number. The latter is incremented with each modification of the entry for reasons that will become clear later. As is usual with version numbers, we assume that enough bits are used for the version numbers that "wraparound" is impossible in practice; see [12] for discussion and justification.) Because there is at most one value that is potentially trapped by guard g at any time, a single location HNDOFF[g] for each guard g is sufficient. To see why, observe that if p needs to hand off v because it is guarded, then the value (if any)—call it w—previously stored in HNDOFF[g] is no longer guarded, so p can pick w up and add it to its set. (Because p attempts to hand off v only if v is in p's set, the Single Location Lemma implies that $v \neq w$.) While this explanation gives the basic idea of our algorithm, it is somewhat oversimplified, as there are various subtle race conditions that must be avoided. We explain how the algorithm deals with these race conditions in more detail below.

To hand v off, p uses a CAS operation to attempt to replace the value previously stored in HNDOFF[g] with v (line 18); this ensures that, upon success, p knows which value it replaced, so it can add that value to its set (line 20). We explain later why it is safe to do so. If the CAS fails due to a concurrent Liberate operation, then p rereads HNDOFF[g] (line 24) and loops around to retry the handoff. There are various conditions under which we break out of this loop and move on to the next guard. (Note in particular that the loop completes after at most three CAS attempts; see lines 13, 22, and 23. Thus our algorithm is wait-free.) We explain later why it is safe to stop trying to hand off v in each of these cases. Before doing so, we explain one more aspect of the algorithm.

As described so far, p picks up a value from HNDOFF[g] only if its value set contains a value that is guarded by guard g. Therefore, without some additional mechanism, it would be quite possible for a value to be stored in HNDOFF[g] and never be picked up from there. This would violate the *value progress* property discussed in Section 2. To avoid this problem, even if p does not need to remove a value from its set, it still picks up the previously handed off value (if any) by replacing it with **null** (see lines 28 through 30).

We now consider each of the ways p can break out of the loop at lines 17 through 26, and explain why it is safe to do so. We first consider the case in which p exits the loop due to a successful CAS at line 18. In this case, as described earlier, p removes v from its set (line 19), adds the previous value in HNDOFF[g] to its set (line 20), and moves on to the next guard (lines 21 and 31). An important part of understanding our algorithm is to understand why it is safe to take the previous value—call it w—of HNDOFF[g] to the next guard. The reason is that we read POST[g] (line 15 or 26) between reading HNDOFF[g] (line 14 or 24) and attempting the CAS at line 18. Because each modification to HNDOFF[g] increments its version number field, it follows that w was in HNDOFF[g] when p read POST[g]. Also, recall that $w \neq v$ in this case. Therefore, when p read POST[g], w was not guarded by g. Furthermore, because w remained in HNDOFF[g] from that moment until the CAS, w cannot become trapped in this interval (because a value can become trapped only while it is injail, and all values in the HNDOFF array and in the sets of Liberate operations are escaping). The same argument explains why it is safe to pick up the value replaced by **null** at line 29.

It remains to consider how p can break out of the loop *without* performing a successful CAS. In each case, p can infer that v is not trapped by g, so it can give up on its attempt to hand off v. If p breaks out of the loop at line 26, then v is not trapped by g at that moment simply because it is not even guarded by g. The other two cases (lines 23 and 25) occur only after a certain number of times around the loop, implying a certain number of failed CAS's.

To see why we can infer that v is not trapped in each of these two cases, consider the timing diagram in Figure 4. (For the rest of this section, we use the notation v_p to indicate the value of

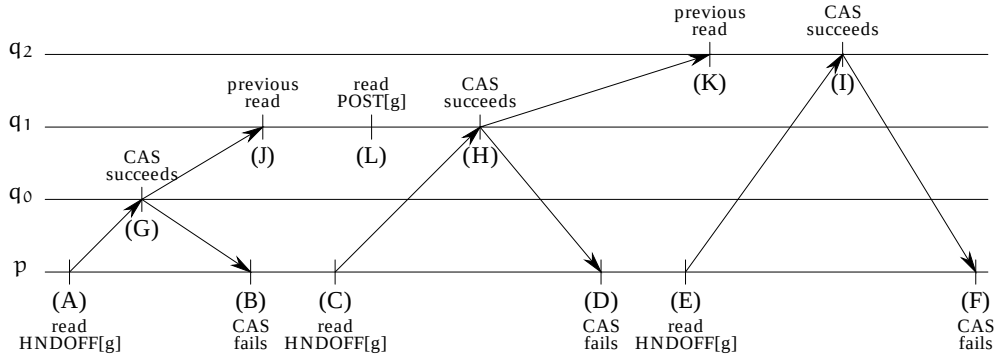

Figure 4: Timing diagram illustrating interesting cases for Pass The Buck.

thread p's local variable v in order to distinguish between the local variables of different threads.) In this diagram, we construct an execution in which p fails its CAS three times. The bottom line represents thread p: at (A), p reads HNDOFF[g] for the first time (line 14); at (B), p's CAS fails; at (C), p rereads HNDOFF[g] at line 24; and so on for (D), (E), and (F). Because p's CAS at (B) fails, some other thread q_0 executing Liberate performed a successful CAS after (A) and before (B); choose one and call it (G). (The arrows between (A) and (G) and between (G) and (B) indicate that we know (G) comes after (A) and before (B).) Similarly, some thread q_1 executes a successful CAS on HNDOFF[g] after (C) and before (D)—call it (H); and some thread q_2 executes a successful CAS on HNDOFF[g] after (E) and before (F)—call it (I). (Note that threads q_0 through q_2 might not be distinct, but there is no loss of generality in treating them as if they were.)

Consider the CAS at (H). Thread q_1's previous read of HNDOFF[g] (at line 14 or line 24)—call it (J)—must come after (G) because every successful CAS increments the version number field of HNDOFF[g]. Similarly, q_2's previous read of HNDOFF[g] before (I)—call it (K)—must come after (H).

We consider two cases. First, suppose (H) is an execution of line 18 by q_1. In this case, between (I) and (H), q_1 read POST[g] $= v_{q_1}$, either at line 15 or at line 26; call this read (L). By the Single Location Lemma, because v_p is in p's set, the read at L implies that v_p was not guarded by g at (L). Therefore, v_p was not trapped by g at (L), which implies that it is safe for p to break out of the loop after (D) in this case (observe that $\text{attempts}_p = 2$ in this case).

For the second case, suppose (H) is an execution of line 29 by thread q_1. In this case, because q_1 is storing **null** instead of a value in its own set, the above argument does not work. However, because p breaks out of the loop at line 25 only if it reads a non-**null** value from HNDOFF[g] at line 24, it follows that if p does so, then *some* successful CAS stored a non-**null** value to HNDOFF[g] at or after (H), and in this case the above argument can be applied to that CAS to show that v_p was not trapped. If p reads **null** at line 24 after (D), then it continues through its next loop iteration.

In this case, there is a successful CAS (I) that comes after (H). Because (H) stored **null** in the current case, no subsequent execution of line 29 by any thread will succeed before the next successful execution of the CAS in line 18 by some thread. To see why, observe that the CAS at line 29 never succeeds while HNDOFF[g] contains **null** (see line 28). Therefore, for (I) to exist, there is a successful execution of the CAS at line 18 by some thread after (H) and at or before (I). Using this CAS, we can apply the same argument as before to conclude that v_p was not trapped. This argument is formalized in an appendix. It is easy to see that our ROP solution is wait-free.

Our algorithm also satisfies the *value progress* property described in Section 2. Briefly, this is because a value cannot remain handed off at a particular guard forever if Liberate is executed enough times. Specifically, a value v is handed off at guard g, then the first Liberate operation to begin processing guard g after v is not trapped by g will ensure that v is picked up and taken to the next guard (or returned from Liberate if g is the last guard), either by that Liberate operation or some concurrent Liberate operation.

As stated earlier, Michael [9] has independently and concurrently developed a solution to a very similar problem. Michael's algorithm buffers to-be-freed values so that it can control the number of values passed to *Scan* (his equivalent of the Liberate operation) at a time. This has the disadvantage that there are usually $O(GP)$ values that could potentially be freed, but are not (where G is the number of "hazard pointers"—the equivalent of guards—and P is the number of participating threads). However, this technique allows him to achieve a nice amortized bound on time spent per value freed. He also has weaker requirements for *Scan* than we have for Liberate; in particular, *Scan* can return some values to the buffer if they cannot yet be freed. This admits a very simple solution that uses only read and write primitives (recall that ours requires CAS), and allows several optimizations. However, it also means that if a thread terminates while values remain in its buffer, then those values will never be freed, so his algorithm does not satisfy the *value progress* property. This is undesirable because a single value might represent a large amount of resources, which would never be reclaimed in this case. The number of such values is bounded by $O(GP)$. We can perform the same optimizations and achieve the same amortized bound under normal operation, while still retaining the *value progress* property (although we would require threads to invoke a special wait-free operation before terminating to achieve this). In this case, our algorithm would perform almost identically to Michael's (with a slight increase in overhead upon thread termination), but would of course share the disadvantages discussed above, except for the lack of value progress.

In [6] we present a dynamic-sized lock-free FIFO queue achieved by applying our ROP solution (without the optimizations discussed above) to the non-dynamic-sized implementation of Michael and Scott [11] together with the non-dynamic-sized freelist of Treiber [13]. Performance experiments presented in that paper show that the overhead of the dynamic-sized FIFO queue over the non-dynamic-sized one of [11] is negligible in the absence of contention, and low in all cases.

Michael has shown how to apply his technique to achieve dynamic-sized implementations of a number of different data structures, including queues, double-ended queues, list-based sets, and hash tables (see [9] for references). Because the interfaces and safety properties of our approaches are almost identical, those results can all be achieved using any ROP solution too. In addition, using the one presented here would allow us to achieve value progress in those implementations. Michael also identified a small number of implementations to which his method is not applicable. In some cases, this may be because Michael's approach is restricted to use a fixed number of hazard pointers per thread; in contrast, ROP solutions provide for dynamic allocation of guards. Furthermore, in [6], we have presented a general methodology based on any ROP solution that can be applied to achieve dynamic-sized versions of these data structures too. This methodology is based on reference counts, and therefore shares the disadvantages thereof, including space and time overhead for reference counts, as well as the need to deal with cyclic garbage.

4 Concluding Remarks

We have defined the Repeat Offender Problem (ROP), and presented one solution to this problem. Such solutions provide a mechanism for supporting memory management in dynamic-sized lock-free data structures. The utility of this mechanism has been demonstrated elsewhere [6], where we present what we believe are the first dynamic-sized lock-free data structures that can continue to reclaim memory even if some threads fail (although Maged Michael [9] has independently and concurrently achieved such implementations, as discussed in Section 3).

By specifying the ROP as an abstract and general problem, we allow for the possibility of using different solutions for different applications and settings, without the need to redesign or reverify the data structure implementations that employ ROP solutions. We have paid particular attention to allowing much of the work of managing dynamically allocated memory to be done concurrently with the application, using additional processors if they are available.

The ideas in this paper came directly from insights gained and questions raised in our work on lock-free reference counting [3]. This further demonstrates the value of research that assumes stronger synchronization primitives than are currently widely supported.

Future work includes exploring other ROP solutions, and applying ROP solutions to the design of other lock-free data structures. It would be particularly interesting to explore the various ways for scheduling Liberate work.

Acknowledgements: We thank Steve Heller, Paul Martin, and Maged Michael for useful feedback, suggestions, and discussions. In particular, Steve Heller suggested formulating ROP to allow the use of "spare" processors for memory management work.

References

[1] J. Anderson and M. Moir. Using local-spin k-exclusion algorithms to improve wait-free object implementations. *Distributed Computing*, 11:1–20, 1997. A preliminary version appeared in *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, 1994, pp. 141-150.

[2] D. Detlefs, C. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. Steele. Even better DCAS-based concurrent deques. In *Proceedings of the 14th International Conference on Distributed Computing*, pages 59–73, 2000.

[3] D. Detlefs, P. Martin, M. Moir, and G. Steele. Lock-free reference counting. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, pages 190–199, 2001.

[4] M. Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University Technical Report STAN-CS-TR-99-1624, Palo Alto, CA, August 1999.

[5] T. Harris. A pragmatic implementation of non-blocking linked lists. In *Proceedings of the 15th International Symposium on Distributed Computing*, October 2001.

[6] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Dynamic-sized lockfree data structures. Technical Report TR-2002-110, Sun Microsystems Laboratories, 2002.

[7] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

[8] N. Lynch and M. Tuttle. An introduction to input/output automata. Technical Report CWI-Quarterly, 2(3), Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1989.

[9] M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual ACM Symposium on the Principles of Distributed Computing*, 2002.

[10] M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing*, pages 267–276, 1996.

[11] M. Michael and M. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[12] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 219–228, 1997.

[13] R. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center, 1986.

[14] J. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 214–22, 1995. See `http://www.cs.sunysb.edu/~valois` for errata.

[15] D. Weaver and T. Germond. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall, Englewood Cliffs, NJ 07632, USA, 1994.

A Formal Proof that PTB Implements ROPlite

In this appendix, we give a formal proof that the Pass The Buck algorithm (PTB) of Section 3 implements *ROPlite*, that is, it solves the Repeat Offender Problem except that it does not provide an IsInJail operation.

To prove that PTB implements *ROPlite*, we first define an I/O automaton model [8] for the PTB algorithm and then show that this automaton implements the ROP automaton in Section 2 (without the IsInJailInv and IsInJailResp actions).

The PTB automaton appears in Figures 5, 6 and 7. It has the same environment actions and output actions as the ROP automaton in Figure 2 except that it does not have the IsInJailInv and IsInJailResp actions because PTB does not implement the IsInJail operation.

The PTB automaton also has internal actions that model the execution of code that implements the operations. Because an action changes the state of the automaton atomically, each action corresponds to code that includes at most one access to shared variables. (Access to local variables always appears atomic because other threads cannot see the effects of such access.) The names of the internal actions indicate the lines of the code that implement them in Figure 3. For $\mathsf{line2\text{-}3}_{\mathrm{p}}$, an occurrence of the action corresponds to a single iteration of the **while** loop. Not all lines of code are modelled by internal actions: Lines 1 and 11 are collapsed into the invocation actions because they access only local variables, and lines 6, 8, 10 and 32 are modelled by the response actions. We discuss the internal actions of Liberate in more detail below.

The PTB automaton specified here can be viewed as the composition of two subautomata, representing the environment and the PTB algorithm, and the state variables defined in Figure 5 can be partitioned between these subautomata. The *environment state variables* are exactly the variables of the ROP automaton. These variables are used to determine when the environment actions are enabled. The *internal PTB state variables* represent state used by the PTB algorithm. These consist of the shared global variables GUARDS, POST, HNDOFF and MAXG; the local variables (including the parameters) of each operation; and a local program counter ln_{p} that records the next line of code to be executed by thread p.

The environment actions of the PTB automaton are identical to their counterparts in the ROP automaton except that they may also set some local variables. The response actions are enabled by the value of the local program counter being equal to the line number corresponding to a return statement. They set the program counter to idle and modify the environment state variables in the same way as the ROP automaton does.[7] Like the response actions, the internal PTB actions are enabled by the local program counter having the appropriate line number, and their effects clauses straightforwardly express the effects of executing the corresponding lines of code.

In addition to the environment assumptions expressed in the ROP automaton, the PTB algorithm assumes that the number of guards simultaneously employed (including those in the midst of being hired or fired) is bounded by MG; the set G of guards is $\{0, \ldots, \mathrm{MG}-1\}$. Failure to satisfy this assumption may result in an attempt at line 2 to write beyond the end of the POST array, possibly overwriting unrelated data structures and causing arbitrary failures. As in the code in Figure 3, we simply assume that this does not happen; we do not specify the behavior of the

[7] That the environment state variables are modified in the same way by both automata is immediate from the definition in all cases except that of PostResp actions. Invariant 1 below establishes that in any reachable state of this automaton, the PostResp actions also update the environment state variables in the same way as in the ROP automaton.

actions

Environment	PTB output	PTB internal
$\mathsf{HireInv}_p()$	$\mathsf{HireResp}_p(g)$	$\mathsf{line2\text{-}3}_p$
$\mathsf{FireInv}_p(g)$	$\mathsf{FireResp}_p()$	$\mathsf{line4}_p$
$\mathsf{PostInv}_p(g, v)$	$\mathsf{PostResp}_p()$	$\mathsf{line5}_p$
$\mathsf{LiberateInv}_p(S)$	$\mathsf{LiberateResp}_p(S)$	$\mathsf{line7}_p$
$\mathsf{Arrest}(v)$		$\mathsf{line9}_p$
		$\mathsf{line12}_p$
		$\mathsf{line13+}_p$
		$\mathsf{line15+}_p$
		$\mathsf{line18+}_p$
		$\mathsf{line22+}_p$
		$\mathsf{line25+}_p$
		$\mathsf{line28+}_p$
		$\mathsf{line31}_p$

state variables

(internal PTB state variables)
- GUARDS: **array**[0..MG-1] **of bool init** all false;
- MAXG: **int init** 0;
- POST: **array**[0..MG-1] **of value init** all **null**;
- HNDOFF: **array**[0..MG-1] **of value_set init** all empty;
- For each thread $p \in P$:
  - ln_p: {idle, 2, 4, 5, 6, 7, 8, 9, 10, 12, 13, 15, 18, 22, 25, 28, 31, 32} **init** idle
  - i_p: **int**
  - v_p: **value**
  - vs_p: **value_set**
  - max_p: **int**
  - $attempts_p$: **value**

(environment state variables)
- For each thread $p \in P$:
  - pc_p: {idle, hire, fire, post(h, v), injail(v), liberate} **init** idle
  - $guards_p$: set of guards **init** empty
- For each value $v \in V$:
  - *status*[v]: {injail, escaping, free} **init** free
- For each guard $g \in G$:
  - *post*[g]: V **init null**
  - *trapping*[g]: **bool init** false
- *numescaping*: **int init** 0

Figure 5: Actions and state variables for the PTB automaton.

HireGuard	HireInv$_p()$
	Pre: $pc_p = \mathsf{idle}$
	Eff: $pc_p \leftarrow \mathsf{hire}$
	$i_p \leftarrow 0$
	$ln_p \leftarrow 2$
	
	line2-3$_p$
	Pre: $ln_p = 2$
	Eff: if $\neg$GUARDS$[i_p]$ then
	GUARDS$[i_p] \leftarrow$ true
	$ln_p \leftarrow 4$
	else
	$i_p \leftarrow i_p + 1$
		line4$_p$
		Pre: $ln_p = 4$
		Eff: $max_p \leftarrow$ MAXG
		if $max_p < i_p$ then
		$ln_p \leftarrow 5$
		else
		$ln_p \leftarrow 6$
		
		line5$_p$
		Pre: $ln_p = 5$
		Eff: if MAXG $= max_p$ then MAXG $\leftarrow i_p$
		$ln_p \leftarrow 4$
		
		HireResp$_p(g)$
		Pre: $ln_p = 6$
		$g = i_p$
		Eff: $ln_p \leftarrow \mathsf{idle}$
		$pc_p \leftarrow \mathsf{idle}$
		$guards_p \leftarrow guards_p \cup \{g\}$

FireGuard	FireInv$_p(g)$ Pre: $pc_p = \mathsf{idle}$ $g \in guards_p$ $post[g] =$ **null** Eff: $pc_p \leftarrow \mathsf{fire}$ $guards_p \leftarrow guards_p - \{g\}$ $i_p \leftarrow g$ $ln_p \leftarrow 7$	line7$_p$ Pre: $ln_p = 7$ Eff: GUARDS$[i_p] \leftarrow$ false $ln_p \leftarrow 8$ FireResp$_p()$ Pre: $ln_p = 8$ Eff: $ln_p = \mathsf{idle}$ $pc_p \leftarrow \mathsf{idle}$
PostGuard	PostInv$_p(g, v)$ Pre: $pc_p = \mathsf{idle}$ $g \in guards_p$ Eff: $pc_p \leftarrow \mathsf{post}(g, v)$ $post[g] \leftarrow$ **null** $trapping[g] \leftarrow$ false $v_p \leftarrow v$ $i_p \leftarrow g$ $ln_p \leftarrow 9$	line9$_p$ Pre: $ln_p = 9$ Eff: POST$[i_p] \leftarrow v_p$ $ln_p \leftarrow 10$ PostResp$_p()$ Pre: $ln_p = 10$ Eff: $ln_p \leftarrow \mathsf{idle}$ $pc_p \leftarrow \mathsf{idle}$ $post[i_p] \leftarrow v_p$ $trapping[i_p] \leftarrow (status[v_p] = \mathsf{injail})$
Arrest	Arrest(v) Pre: $status[v] = \mathsf{free}$ $v \neq$ **null** Eff: $status[v] \leftarrow \mathsf{injail}$ for all g such that $post[g] = v$, $trapping[g] \leftarrow$ true	

Figure 6: Transitions for the PTB automaton, part 1: All but Liberate.

Liberate

LiberateInv$_p(S)$
Pre: $pc_p = \text{idle}$
  for all $v \in S$,
    $status[v] = \text{injail}$
    $v \neq \textbf{null}$
Eff: $pc_p \leftarrow \text{liberate}$
  $numescaping \leftarrow numescaping + |S|$
  for all $v \in S$, $status[v] \leftarrow \text{escaping}$
  $vs_p \leftarrow S$
  $i_p \leftarrow 0$
  $ln_p \leftarrow 12$

line12$_p$
Pre: $ln_p = 12$
Eff: if $i_p \leq \text{MAXG}$ then
    $ln_p \leftarrow 13$
  else
    $ln_p \leftarrow 32$

line13-14$_p$
Pre: $ln_p = 13$
Eff: $attempts_p \leftarrow 0$
  $h_p \leftarrow \text{HNDOFF}[i_p]$
  $ln_p \leftarrow 15$

line15-17$_p$
Pre: $ln_p = 15$
Eff: $v_p \leftarrow \text{POST}[i_p]$
  if $v_p \neq \textbf{null} \wedge v_p \in vs_p$ then
    $ln_p \leftarrow 18$
  else
    $ln_p \leftarrow 28$

line28-30$_p$
Pre: $ln_p = 28$
Eff: if $h_p.val \neq \textbf{null} \wedge h_p.val \neq v_p$
      $\wedge \text{HNDOFF}[i_p] = h_p$ then
    $\text{HNDOFF}[i_p] \leftarrow \langle \textbf{null}, h_p.ver + 1 \rangle$
    $vs_p \leftarrow vs_p \cup \{h_p.val\}$
  $ln_p \leftarrow 31$

line18-21$_p$
Pre: $ln_p = 18$
Eff: if $\text{HNDOFF}[i_p] = h_p$ then
    $\text{HNDOFF}[i_p] \leftarrow \langle v_p, h_p.ver + 1 \rangle$
    $vs_p \leftarrow vs_p - \{v_p\}$
    if $h_p.val \neq \textbf{null}$ then $vs_p \leftarrow vs_p \cup \{h_p.val\}$
    $ln_p \leftarrow 31$
  else
    $ln_p \leftarrow 22$

line22-24$_p$
Pre: $ln_p = 22$
Eff: $attempts_p \leftarrow attempts_p + 1$
  if $attempts_p = 3$ then
    $ln_p \leftarrow 31$
  else
    $h_p \leftarrow \text{HNDOFF}[i_p]$
    $ln_p \leftarrow 25$

line25-26$_p$
Pre: $ln_p = 25$
Eff: if $(attempts_p = 2 \wedge h_p.val \neq \textbf{null})$
      $\vee\, v_p \neq \text{POST}[i_p]$ then
    $ln_p \leftarrow 31$
  else
    $ln_p \leftarrow 18$

line31$_p$
Pre: $ln_p = 31$
Eff: $i_p \leftarrow i_p + 1$
  $ln_p \leftarrow 12$

LiberateResp$_p(S)$
Pre: $ln_p = 32$
  $S = vs_p$
Eff: $ln_p \leftarrow \text{idle}$
  $pc_p \leftarrow \text{idle}$
  $numescaping \leftarrow numescaping - |S|$
  for all $v \in S$, $status[v] \leftarrow \text{free}$

Figure 7: Transitions for the PTB automaton, continued: Liberate.

automaton when $i_p \geq MG$ in the execution of either line2-3$_p$ or line7$_p$.

We now prove several invariants and lemmas about the PTB automaton. Of these, Invariants 1, 2, 12 and 18 are the ones actually used in the proof of Theorem 1, which says that PTB implements *ROPlite*. The rest are used to prove Invariants 12 and 18. Informally, Invariant 1 says that pc_p has the appropriate value when p is executing a PTB operation. Invariant 2 says that every guard is employed by at most one thread, and that if g is employed then GUARDS[g] = *true*.[8] Invariant 12 says that the escaping values are those that appear in the HNDOFF array or in the vs_p set of some thread p with pc_p = liberate, and that each escaping value appears in exactly one of these locations. Invariant 18 says that if a guard g traps a value v and v is in the vs_p set of some thread executing Liberate, then that thread has not yet finished processing g.

We prove most invariants by induction on the number of transitions in an execution. That is, we prove that the invariant holds in the initial state and that for each reachable state s, if the invariant holds in s and $s \xrightarrow{a} s'$ then the invariant holds in s'. For many of the invariants, checking this is straightforward, and we omit the details of the proof. Because we assume s (and thus s') is reachable, we can assume the earlier invariants hold in s and s' in the proofs of later invariants.

Invariant 1 If $ln_p \in \{2,4,5,6\}$ then pc_p = hire. If $ln_p \in \{7,8\}$ then pc_p = fire. If $ln_p \in \{9,10\}$ then pc_p = post(i_p, v_p). If $ln_p \in \{12,13,15,18,22,25,28,31,32\}$ then pc_p = liberate.

Proof: Straightforward by induction. ■

Invariant 2 Every guard g is in exactly one of the following sets: $guards_p$ for some p, $\{i_p\}$ for some p with $ln_p \in \{4,5,6,7\}$, or $\{g : \neg GUARDS[g]\}$.

Proof: This invariant holds initially because $guards_p$ is empty and ln_p = idle for all p and $\neg GUARDS[g]$ for all g. Suppose that the invariant holds in s and that $s \xrightarrow{a} s'$. Let S_p be $\{i_p\}$ if $ln_p \in \{4,5,6,7\}$ and $\emptyset$ otherwise. The only actions that change any of the sets listed in the invariant are line2-3$_p$ for some p with $\neg s.GUARDS[s.i_p]$, HireResp$_p$(g) and FireInv$_p$(g) for some p and g, and line7$_p$ for some p. In each case, one guard is removed from one set and added to another, so the invariant is preserved.

For line2-3$_p$ with $\neg s.GUARDS[s.i_p]$, $s.i_p$ is removed from $\{g : \neg GUARDS[g]\}$ and added to S_p.

For HireResp$_p$(g), $g = s.i_p$, $s.ln_p = 6$ and $s'.ln_p$ = idle, so g is removed from S_p and added to $guards_p$.

For FireInv$_p$(g), g is removed from $guards_p$ and added to S_p.

Finally, for line7$_p$, $s.i_p$ is removed from S_p and added to $\{g : \neg GUARDS[g]\}$. ■

Invariant 3 For all p, $ln_p \in \{9,10\} \implies i_p \in guards_p$.

Proof: Straightforward by induction. ■

Invariant 4 For all p and q, if $ln_p \in \{9,10\}$ and $ln_q \in \{9,10\}$ then $i_p \neq i_q$.

Proof: Immediate from Invariants 2 and 3. ■

[8] To make it provable by induction, the actual invariant is strengthened slightly. This is also true for Invariant 18.

Invariant 5 For all p, $ln_p = 10 \implies \text{POST}[i_p] = v_p$.

Proof: Straightforward by induction with Invariant 4. ■

Invariant 6 For all g,

$$post[g] = \begin{cases} \textbf{null} & \text{if } \exists p(ln_p \in \{9,10\} \wedge i_p = g) \\ \text{POST}[g] & \text{otherwise} \end{cases}$$

Proof: Straightforward by induction with Invariants 1, 4 and 5. ■

Invariant 7 For all p, $ln_p = 5 \implies max_p < i_p$.

Proof: Straightforward by induction. ■

Invariant 8 For all p, $ln_p \in \{2,4,5\} \implies i_p \geq 0$.

Proof: Straightforward by induction. ■

Invariant 9 For all g and p, if $g \in guards_p$ or $ln_p = 6 \wedge i_p = g$ then $0 \leq g \leq \text{MAXG}$.

Proof: Straightforward by induction with Invariants 7 and 8. ■

Invariant 10 For all g, if $post[g] \neq \textbf{null}$ then $0 \leq g \leq \text{MAXG}$.

Proof: Straightforward by induction with Invariants 3, 7 and 9. ■

Invariant 11 For all p, if $ln_p \in \{18, 22, 25\}$ then $v_p \in vs_p$.

Proof: Straightforward by induction. ■

Invariant 12 Every non-**null** value is in exactly one of the following sets: $\{\text{HNDOFF}[g].val\}$ for some g, vs_p for some p with $pc_p = \text{liberate}$, and $\{v : status[v] \neq \text{escaping}\}$.

Proof: This invariant holds initially because $\text{HNDOFF}[g].val = \textbf{null}$ for all g, $pc_p = \text{idle}$ for all p, and $status[v] = \text{injail} \neq \text{escaping}$ for all v. Suppose that s is a reachable state in which the invariant holds, and that $s \xrightarrow{a} s'$. Let $HS_g = \{\text{HNDOFF}[g].val\}$ and VS_p be vs_p if $pc_p = \text{liberate}$ and $\emptyset$ otherwise. With Invariant 1, it is easy to see that the only actions that change any of the sets in the invariant are $\text{LiberateInv}_p(S)$ and $\text{LiberateResp}_p(S)$ for some p and S, line18-21_p for some p with $s.\text{HNDOFF}[s.i_p] = s.h_p$, and line28-30_p for some p with $s.h_p.val \notin \{\textbf{null}, s.v_p\}$ and $s.\text{HNDOFF}[s.i_p] = s.h_p$.

For $\text{LiberateInv}_p(S)$, every value in S is removed from $\{v : status[v] \neq \text{escaping}\}$ and added to VS_p.

For $\text{LiberateResp}_p(S)$, $s.pc_p = \text{liberate}$ by Invariant 1, so $S = VS_p$. Thus, every value in S is removed from VS_p and added to $\{v : status[v] \neq \text{escaping}\}$.

For $\mathsf{line18\text{-}21}_p$ with $s.\mathrm{HNDOFF}[s.i_p] = s.h_p$, values other than $s.v_p$ and $s.h_p$ are not added to or removed from any of the sets. By Invariant 11, $s.v_p \in s.vs_p$ (because $s.ln_p = 18$), so $s.v_p$ is removed from VS_p and added to $HS_{s.i_p}$. If $s.h_p.val \neq \textbf{null}$ then by the inductive hypothesis, $s.h_p \notin s.vs_p$ and so $s.h_p$ is removed from $HS_{s.i_p}$ and added to VS_p.

For $\mathsf{line28\text{-}30}_p$ with $s.h_p.val \notin \{\textbf{null}, s.v_p\}$ and $s.\mathrm{HNDOFF}[s.i_p] = s.h_p$, by the inductive hypothesis, $s.h_p.val \notin s.vs_p$ and so $s.h_p.val$ is removed from $HS_{s.i_p}$ and added to VS_p.

Thus, the invariant that each value is in exactly one of the sets is preserved. ■

Invariant 13 For all p, if $ln_p \in \{15, 18, 22, 25, 28, 31\}$ then $h_p.ver \leq \mathrm{HNDOFF}[i_p].ver$.

Proof: Straightforward by induction. ■

Lemma 1 For all p and g, if s is a reachable state in which $s.i_p = g$ and $s.h_p \neq s.\mathrm{HNDOFF}[g]$, and $s \xrightarrow{\alpha} s'$, then $s'.h_p \neq s'.\mathrm{HNDOFF}[g]$ or $\alpha \in \{\mathsf{line13\text{-}14}_p, \mathsf{line22\text{-}24}_p\}$.

Proof: Fix p and g. If $\alpha \notin \{\mathsf{line13\text{-}14}_p, \mathsf{line22\text{-}24}_p\}$ then $s'.h_p = s.h_p$ and $s'.\mathrm{HNDOFF}[g] = s.\mathrm{HNDOFF}[g]$, unless there is some q with $s.i_q = g$, $s.h_q = s.\mathrm{HNDOFF}[g]$ and either $\alpha = \mathsf{line18\text{-}21}_q$ or $\alpha = \mathsf{line28\text{-}30}_q$ and $s.h_q.val \notin \{\textbf{null}, s.v_q\}$. If there is such a q then $s'.\mathrm{HNDOFF}[g].ver = s.\mathrm{HNDOFF}[g].ver + 1$, and by Invariant 13, $s'.h_p.ver = s.h_p.ver \leq s.\mathrm{HNDOFF}[g].ver$, so $s'.h_p \neq s'.\mathrm{HNDOFF}[g]$. ■

Lemma 2 For all g and $v \neq \textbf{null}$, if s is reachable, $s \xrightarrow{\alpha} s'$, and $s'.post[g] = v$ and $s'.trapping[g]$ then either $s.post[g] = v$ and $s.trapping[g]$ or $s'.status[v] = \mathsf{injail}$.

Proof: Fix g and v. If $s'.post[g] = v$ and $s'.trapping[g]$ and $s.post[g] \neq v \vee \neg s.trapping[g]$ then either $\alpha = \mathsf{PostResp}_p()$ for some p with $s.i_p = g$, $s.v_p = v$ and $s.status[v] = \mathsf{injail}$, or $\alpha = \mathsf{Arrest}(v)$ with $s.post[g] = v$. In either case, $s'.status[v] = \mathsf{injail}$. ■

Invariant 14 For all g, $v \neq \textbf{null}$ and p, if $post[g] = v = \mathrm{HNDOFF}[g].val$, $trapping[g]$, $i_p = g$ and $h_p = \mathrm{HNDOFF}[g]$ then $ln_p \neq 18$ and $ln_p = 28 \implies v_p = v$.

Proof: Fix g, v and p. This invariant holds initially because $trapping[g] = false$. Suppose it holds in a reachable state s, $s \xrightarrow{\alpha} s'$, $s'.post[g] = v = s'.\mathrm{HNDOFF}[g].val$, $s'.trapping[g]$, and $s'.h_p = s'.\mathrm{HNDOFF}[g]$.

If $s.i_p \neq g$ then $s'.i_p \neq g$ or $s'.ln_p \notin \{18, 28\}$, so the invariant holds in s'.

If $s.i_p = g$ and $s.h_p \neq s.\mathrm{HNDOFF}[g]$ then by Lemma 1, because $s'.h_p = s'.\mathrm{HNDOFF}[g]$, we have $\alpha = \{\mathsf{line13\text{-}14}_p, \mathsf{line22\text{-}24}_p\}$, so $s'.ln_p \notin \{18, 28\}$.

If $s.\mathrm{HNDOFF}[g].val \neq v$ then $\alpha = \mathsf{line18\text{-}21}_q$ for some q with $s.i_q = g$ and $s.h_q = s.\mathrm{HNDOFF}[g]$. If $s'.ln_p \notin \{18, 28\}$ then the invariant holds in s'. Otherwise, $s'.ln_p \in \{18, 28\}$, so $s.ln_p \in \{18, 28\}$ (because $\alpha = \mathsf{line18\text{-}21}_q$, so $p \neq q$). In the latter case, by Invariant 13, $s.h_p.ver \leq s.\mathrm{HNDOFF}[g].ver$, and because $s'.\mathrm{HNDOFF}[g].ver = s.\mathrm{HNDOFF}[g].ver + 1$ and $s'.h_p.ver = s.h_p.ver$, we have $s'.h_p \neq s'.\mathrm{HNDOFF}[g]$, so the invariant holds in s'.

Otherwise, $s.\mathrm{HNDOFF}[g].val = v$, $s.i_p = g$ and $s.h_p = s.\mathrm{HNDOFF}[g]$. By Invariant 12, $s'.status[v] = \mathsf{escaping} \neq \mathsf{injail}$, so by Lemma 2, $s.post[g] = v$ and $s.trapping[g]$. Thus, by the inductive hypothesis, $s.ln_p \neq 18$ and $s.ln_p = 28 \implies s.v_p = v$. If $\alpha \notin \{\mathsf{line15\text{-}17}_p, \mathsf{line25\text{-}26}_p\}$

then $s'.ln_p \neq 18$ and $s'.ln_p = 28 \implies s'.v_p = v$, so the invariant holds in s'. Otherwise, $s.ln_p \in \{15, 25\}$, so by Invariant 1, $s.pc_p = \mathsf{liberate}$, and by Invariant 12, $v \notin s.vs_p$. Also, by Invariant 6, $s.POST[g] = s.post[g] = v$.

If $a = \mathsf{line15\text{-}17}_p$ then $s'.v_p = s.POST[g] = v$ and $s'.ln_p = 28$, so the invariant holds in s'.

If $a = \mathsf{line25\text{-}26}_p$ then by Invariant 11, $s.v_p \in s.vs_p$, so $s.v_p \neq v = s.POST[g]$. Thus, $s'.ln_p = 31$ and the invariant holds in s'. ■

Lemma 3 For all g, $v \neq$ **null** and p, if s is reachable, $s \xrightarrow{a} s'$, $s.post[g] = v = s.HNDOFF[g].val$, $s.trapping[g]$, $s.i_p = g$ and $s.ln_p \in \{18, 28\}$ then $v \notin s'.vs_p$.

Proof: Because $v = s.HNDOFF[g].val$, by Invariants 1 and 12, $v \notin s.vs_p$. Either $s'.vs_p = s.vs_p$, and so $v \notin s'.vs_p$, as required, or $a \in \{\mathsf{line18\text{-}21}_p, \mathsf{line28\text{-}30}_p\}$ and $s.h_p = s.HNDOFF[s.i_p]$. In the latter case, because $s.i_p = g$ and $s.ln_p \in \{18, 28\}$, by Invariant 14, we have $s.ln_p \neq 18$ and $s.v_p = v \notin s.vs_p$. Thus, $a = \mathsf{line28\text{-}30}_p$, and because $s.h_p.val = v$, $v \notin s.vs_p = s'.vs_p$. ■

Lemma 4 For all g and p, if $s \xrightarrow{a} s'$, $s'.i_p = g$ and $s'.ln_p \in \{15, 18, 22, 25\}$, then $s.i_p = g$ and $s.vs_p = s'.vs_p$.

Proof: Straightforward by inspection. ■

Invariant 15 For all g, $v \neq$ **null**, p and $q \neq p$, if $post[g] = v \in vs_p$, $trapping[g]$, $i_p = i_q = g$, $ln_p \in \{15, 18, 22, 25\}$, $ln_q = 18$ and $h_q = HNDOFF[g]$ then $h_p = HNDOFF[g]$, $attempts_p = 0$ and $ln_p \neq 22$.

Proof: Fix g, v, p and q. This invariant holds initially because $trapping[g] = false$. Suppose that it holds in a reachable state s, $s \xrightarrow{a} s'$, and $s'.post[g] = v \in s'.vs_p$, $s'.trapping[g]$, $s'.i_p = s'.i_q = g$, $s'.ln_p \in \{15, 18, 22, 25\}$, $s'.ln_q = 18$ and $s'.h_q = s'.HNDOFF[g]$. By Lemma 4, $s.i_p = s.i_q = g$ and $v \in s.vs_p = s'.vs_p$.

If $s.ln_p \notin \{15, 18, 22, 25\}$ then $a = \mathsf{line13\text{-}14}_p$, so $s'.h_p = s.HNDOFF[s.i_p] = s'.HNDOFF[g]$, $s'.attempts_p = 0$ and $s'.ln_p = 15 \neq 22$, as required.

Otherwise, $s.ln_p \in \{15, 18, 22, 25\}$, so by Invariant 1, $s.pc_p = \mathsf{liberate}$, and by Invariant 12, $s'.status[v] = \mathsf{escaping} \neq \mathsf{injail}$ (because $v \in s'.vs_p$), so by Lemma 2, $s.post[g] = v$ and $s.trapping[g]$. Because $s'.ln_q = 18$, we have $a \notin \{\mathsf{line13\text{-}14}_q, \mathsf{line22\text{-}24}_q\}$, so by Lemma 1, $s.h_q = s.HNDOFF[g]$. Also, $s.ln_q = 18$ because otherwise, either $a = \mathsf{line15\text{-}17}_q$ and $s.POST[g] \in s.vs_q$ or $a = \mathsf{line25\text{-}26}_q$ and $s.v_q = s.POST[g]$, which, by Invariant 11, also implies $s.POST[g] \in s.vs_q$. However, both these cases are impossible because, by Invariant 6, $s.POST[g] = v$, and by Invariants 1 and 12, $v \notin s.vs_q$ because $v \in s.vs_p$ and $p \neq q$.

Thus, by the inductive hypothesis, $s.h_p = s.HNDOFF[g]$, $s.attempts_p = 0$ and $s.ln_p \neq 22$. So $s'.attempts_p = s.attempts_p = 0$, and $s'.h_p = s.h_p = s.HNDOFF[g]$. Because $s.ln_q = s'.ln_q = 18$, we have $s'.h_q = s.h_q$, thus, $s'.h_p = s.HNDOFF[g] = s.h_q = s'.h_q = s'.HNDOFF[g]$. Finally, $s'.ln_p \neq 22$ because $s.ln_p \neq 22$ and $s.HNDOFF[s.i_p] = s.h_p$. Thus, the invariant holds in s'. ■

Invariant 16 For all p, $ln_p = 15 \implies attempts_p = 0$.

Proof: Straightforward by induction. ■

Invariant 17 For all p, if $ln_p = 22$ then $h_p \neq \text{HNDOFF}[i_p]$.

Proof: Straightforward by induction with Lemma 1. ■

Invariant 18 For all g and $v \neq \textbf{null}$, if *post*[g] $= v$ and *trapping*[g] then

1. for all $h > g$, $v \neq \text{HNDOFF}[h].val$;
2. for all p, if $pc_p = \mathsf{liberate}$ and $v \in vs_p$ then $ln_p \in \{12, 13, 15, 18, 22, 25, 28, 31\}$ and $i_p \leq g$;
3. for all p, if $pc_p = \mathsf{liberate}$, $v \in vs_p$, $i_p = g$ and $ln_p = \{18, 22, 25, 28, 31\}$ then
   - $v_p = v$,
   - $ln_p \notin \{28, 31\}$,
   - $\text{attempts}_p = 2 \implies h_p = \text{HNDOFF}[g] \wedge h_p.val = \textbf{null}$,
   - $\text{attempts}_p = 1 \implies h_p = \text{HNDOFF}[g] \vee \text{HNDOFF}[g].val = \textbf{null}$.

Proof: Fix g and v. This invariant holds initially because *trapping*[g] = *false*. Suppose it holds in a reachable state s, $s \xrightarrow{a} s'$, and $s'.post[g] = v$ and $s'.trapping[g]$. By Lemma 2, either $s.post[g] = v$ and $s.trapping[g]$ or $s'.status[v] = \mathsf{injail} \neq \mathsf{escaping}$. In the latter case, by Invariant 12, $v \neq s'.\text{HNDOFF}[h].val$ for all h and $v \notin s'.vs_p$ for any p such that $pc_p = \mathsf{liberate}$, so this invariant holds in s'. For the rest of the proof, we consider the case in which $s.post[g] = v$ and $s.trapping[g]$.

For the first clause, fix $h > g$. By the inductive hypothesis, $v \neq s.\text{HNDOFF}[h].val$, and thus, $v \neq s'.\text{HNDOFF}[h].val$ unless $a = \mathsf{line18\text{-}21}_p$ for some p with $s.i_p = h$, $s.v_p = v$ and $s.h_p = s.\text{HNDOFF}[h]$. However, if this action is enabled in s then $s.ln_p = 18$, so by Invariant 1, $s.pc_p = \mathsf{liberate}$, and by the inductive hypothesis (second clause), either $v \notin s.vs_p$ or $s.i_p \leq g$. The first possibility contradicts Invariant 11 because $s.v_p = v$; the second contradicts $s.i_p = h > g$. Thus, $v \neq s'.\text{HNDOFF}[h].val$.

For the second and third clauses, fix p. If $s.pc_p \neq \mathsf{liberate}$ then either $s'.pc_p \neq \mathsf{liberate}$, in which case these clauses hold in s', or $a = \mathsf{LiberateInv}(S)$ for some S. In the latter case, $s'.ln_p = 12$ and, by Invariant 10, $s'.i_p = 0 \leq g$, so both clauses hold in s'.

If $s.pc_p = \mathsf{liberate}$ and $v \notin s.vs_p$ then either $v \notin s'.vs_p$, in which case both clauses hold in s', or $a \in \{\mathsf{line18\text{-}21}_p, \mathsf{line28\text{-}30}_p\}$, $s.\text{HNDOFF}[s.i_p] = s.h_p$ and $s.h_p.val = v$. In the latter case, $s'.ln_p = 31$ and $s'.i_p = s.i_p \leq g$ because by the inductive hypothesis (first clause), $s.\text{HNDOFF}[h].val \neq v = s.\text{HNDOFF}[s.i_p]$ for all $h > g$. Furthermore, because $s.ln_p \in \{18, 28\}$, if $s.i_p = g$ then by Lemma 3, $v \notin s'.vs_p$. So both clauses hold in s'.

It remains only to check the case in which $s.pc_p = \mathsf{liberate}$ and $v \in s.vs_p$. In this case, by the inductive hypothesis (second clause), $s.ln_p \in \{12, 13, 15, 18, 22, 25, 28, 31\}$ and $s.i_p \leq g$. Thus, $s'.ln_p \in \{12, 13, 15, 18, 22, 25, 28, 31\}$ and $s'.i_p \leq g$ unless either $a = \mathsf{line12}_p$ and $s.i_p > s.\text{MAXG}$ or $a = \mathsf{line31}_p$ and $s.i_p = g$. The first case is impossible because, by Invariant 10, $s.\text{MAXG} \geq g \geq s.i_p$; the second case is impossible because $s.i_p = g \implies ln_p \neq 31$ by the third clause of the inductive hypothesis. So the second clause of the invariant holds in s'.

For the third clause, if $s.i_p \neq g$ then $s'.i_p \neq g$ unless $a = \mathsf{line31}_p$, in which case $s'.ln_p = 12$, so the third clause holds in s'. If $s.i_p = g$ and $s.ln_p \notin \{18, 22, 25, 28, 31\}$ then $s'.ln_p \notin \{18, 22, 25, 28, 31\}$ unless $a = \mathsf{line15\text{-}17}_p$. In this case, by Invariant 6, $s.\text{POST}[s.i_p] = s.post[s.i_p] = v \in s.vs_p$, so $s'.v_p = v$, $s'.ln_p = 18$ and, by Invariant 16, $s'.\text{attempts}_p = s.\text{attempts}_p = 0$. So the invariant holds in s'.

If $s.i_p = g$ and $s.ln_p \in \{18, 22, 25, 28, 31\}$ then, by the inductive hypothesis (third clause), so $s.ln_p \notin \{28, 31\}$ and $s.v_p = v$. The only actions we need to consider are $\mathsf{line18\text{-}21}_p$, $\mathsf{line22\text{-}24}_p$, $\mathsf{line25\text{-}26}_p$, and $\mathsf{line18\text{-}21}_q$ and $\mathsf{line28\text{-}30}_q$ for some $q \neq p$ with $s.i_q = g$ and $s.h_q = s.\text{HNDOFF}[g]$. We check for each of these actions that the third clause of the invariant holds in s'.

For $a = \mathsf{line18\text{-}21}_p$, we have two cases. If $s.h_p = s.\text{HNDOFF}[g]$ then by Invariant 11, $v = s.v_p \in s.vs_p$, and by Invariants 1 and 12, $s.h_p.val \neq v$, so $v \notin s'.vs_p$, and the third clause holds in s'. Otherwise, $s'.ln_p = 22$ and the third clause is preserved by a.

For $a = \mathsf{line22\text{-}24}_p$, by Invariant 17 and the inductive hypothesis, $s.\text{attempts}_p \neq 2$, and $s.\text{attempts}_p = 1 \implies s.\text{HNDOFF}[g].val = \textbf{null}$. Thus, $s'.ln_p = 25$, $s'.v_p = s.v_p = v$, $s'.h_p = s.\text{HNDOFF}[g] = s'.\text{HNDOFF}[g]$ and $s'.\text{attempts}_p = 2 \implies s'.h_p.val = \textbf{null}$.

For $a = \mathsf{line25\text{-}26}_p$, by Invariant 6 and the inductive hypothesis, $s.v_p = v = s.post[s.i_p] = s.\text{POST}[s.i_p]$ and $s.\text{attempts}_p = 2 \implies s.h_p.val = \textbf{null}$. Thus, $s'.ln_p = 18$ and the third clause is preserved by a.

For $a = \mathsf{line18\text{-}21}_q$ for some $q \neq p$ with $s.i_q = g$ and $s.h_q = s.\text{HNDOFF}[g]$, by Invariant 15, $s'.\text{attempts}_p = s.\text{attempts}_p = 0$. Also, $s'.v_p = s.v_p = v$ and $s'.ln_p = s.ln_p \notin \{28, 31\}$.

For $a = \mathsf{line28\text{-}30}_q$ for some $q \neq p$ with $s.i_q = g$ and $s.h_q = s.\text{HNDOFF}[g]$, we consider three cases. If $s.\text{attempts}_p = 2$ then $s.h_q.val = s.\text{HNDOFF}[g].val = \textbf{null}$ by the inductive hypothesis, so a preserves the third clause. If $s.\text{attempts}_p = 1$ and $s.h_q.val \in \{\textbf{null}, s.v_q\}$ then again, a preserves the third clause. If $s.\text{attempts}_p = 1$ and $s.h_q.val \notin \{\textbf{null}, s.v_q\}$ then $s'.\text{attempts}_p = s.\text{attempts}_p = 1$ and $s'.\text{HNDOFF}[g].val = \textbf{null}$. ■

We now prove the main theorem, which says that PTB implements *ROPlite* (assuming a well-behaved environment). This proof uses a special case of the *simulation relation*, or *refinement*, method [8].

Theorem 1 The PTB automaton (including the environment part) implements the ROP automaton (including the environment part).

Proof: We can prove that one automaton A implements another automaton B with the same input and output actions (but different internal actions) by giving a function f from the states of A to the states of B that satisfies the following properties:

1. If s is an initial state of A then $f(s)$ is an initial state of B.
2. If s is a reachable state of A and $s \xrightarrow{a} s'$ then $f(s) \xrightarrow{a} f(s')$ if a is external and $f(s') = f(s)$ if a is internal.

The existence of such a function, called a *refinement* or a *simulation*, implies that A implements B [8].[9] Because the state variables of the ROP automaton are exactly the environment state variables of the PTB automaton, we simply use the function that maps states of the PTB automaton to the states of the ROP automaton with identical values for those state variables. The first property above is satisfied because the environment state variables are initialized to the same values in both automata.

Suppose that s is a reachable state of the PTB automaton and $s \xrightarrow{a} s'$. If a is an internal action, then $f(s') = f(s)$ because the internal actions do not modify any of the environment state variables.

[9] This is a special case of refinements and simulations, but it is sufficient for our purposes.

If a is an environment action then the precondition of a is identical in the two automata and the environment state variables are updated in the same way by both automata, so $f(s) \xrightarrow{a} f(s')$, as required.

For the PTB output actions, we need to argue that a is enabled in $f(s)$ and that both automata update the environment state variables in the same way. For HireResp, FireResp and LiberateResp actions, the latter is obvious because the statements of the effects clause that affect the environment state variables are identical.

For $\mathsf{HireResp}_p(g)$, $s.ln_p = 6$ and $s.i_p = g$, so by Invariant 1, $s.pc_p = \mathsf{hire}$, and by Invariant 2, $g \notin \bigcup_q s.guards_q$.

For $\mathsf{FireResp}_p()$, $s.ln_p = 8$, so by Invariant 1, $s.pc_p = \mathsf{fire}$.

For $\mathsf{LiberateResp}_p(S)$, $s.ln_p = 32$ and $S = s.vs_p$, so $s.pc_p = \mathsf{liberate}$ by Invariant 1, $s.status[v] = \mathsf{escaping}$ for all $v \in s.vs_p = S$ by Invariant 12, and for all g and $v \neq \textbf{null}$ such that $s.post[g] = v$ and $s.trapping[g]$, we have $v \notin s.vs_p = S$ by Invariant 18 (second clause).

For $\mathsf{PostResp}_p()$ for some p, $s.ln_p = 10$, so by Invariant 1, $s.pc_p = \mathsf{post}(s.i_p, s.v_p)$. Thus, $\mathsf{PostResp}_p()$ is enabled in $f(s)$, and the environment state variables are updated in the same way in both automata, so $f(s) \xrightarrow{a} f(s')$, as required. ■

About the Authors

Maurice Herlihy is a professor at the Brown University Computer Science Department. Before joining Brown in 1993, he was a member of research staff at Digital's Cambridge Research Laboratory, and before that a faculty member at Carnegie Mellon University's School of Computer Science. His interests include practical and theoretical aspects of distributed systems. He received a Ph.D. in Computer Science from MIT, and an A.B. in Mathematics from Harvard.

Victor Luchangco works in the Scalable Synchronization Group of Sun Microsystems Laboratories. His research focuses on algorithms and mechanisms to support concurrent programming on large-scale distributed systems. Although he is a theoretician by disposition and training, he is also interested in practical aspects of computing; he would like to design mechanisms that people will actually use. He also wants to explore how to make proofs for concurrent systems easier, both by changing how people design these systems and by using tools to aid in formal verification. He received an Sc.D. in Computer Science from MIT in 2001, with a dissertation on models for weakly consistent memories.

Mark Moir received the B.Sc. (Hons.) degree in Computer Science from Victoria University of Wellington, New Zealand in 1988, and the Ph.D. degree in Computer Science from the University of North Carolina at Chapel Hill, USA in 1996. From August 1996 until June 2000, he was an assistant professor in the Department of Computer Science at the University of Pittsburgh. In June 2000, he joined Sun Microsystems Laboratories, where he is now the Principal Investigator of the Scalable Synchronization Research Group.

Dr. Moir's main research interests concern practical and theoretical aspects of concurrent, distributed, and real-time computing. His current research focuses on mechanisms for non-blocking synchronization in shared-memory multiprocessors.