Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 254 (2009) 123–141
www.elsevier.com/locate/entcs

Formal Verification of a Reader-Writer Lock Implementation in C

Mark A. Hillebrand[1,2] Dirk C. Leinenbach[1,3]

German Research Center for Artificial Intelligence (DFKI)
Saarbrücken, Germany

Abstract

We present the formal verification of a reader-writer lock implementation, which is a widely used synchronization primitive in multithreaded code. Specifications are given at the level of C code in the annotation language of Microsoft's Verifying C Compiler (VCC); VCC generates and discharges all verification conditions automatically. In addition to lock acquisition and release, we also deal with lock initialization. To accommodate different lock initialization patterns in client code, initialization is modeled in two phases. This work is part of a larger effort to specify and verify Microsoft's hypervisor Hyper-V at the code level in the context of the Verisoft XT project. Our results have been successfully transferred to the real lock implementation of Hyper-V and successfully used in the verification of client code.

Keywords: code verification, systems verification, concurrency, lock primitives

1 Introduction

Synchronization primitives like locks and semaphores provide important means of concurrency control in multithreaded code. Where multithreaded code at the application level relies on the implementation of synchronization primitives from elsewhere, multithreaded operating systems (and thread libraries) implement their own synchronization primitives. Their correctness is both critical to overall system correctness and non-trivial, since such code contains races and is

[1] Work funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft XT project under grant 01 IS 07 008.
[2] Email: mah@dfki.de
[3] Email: dirk.leinenbach@dfki.de

doi:10.1016/j.entcs.2009.09.063

sensitive to memory models and compiler optimizations. In this paper we report on the specification and verification of a C implementation of reader-writer locks, which are a widely used synchronization primitive.

The remainder of this paper is structured as follows. In Section 2 we give a short overview on related work in the area of concurrency and lock verification. In Section 3 we give an introduction to the annotation language and methodology of the Verifying C Compiler (VCC) developed at Microsoft, which we use to annotate and verify our lock implementation. In Section 4 we present the reader-writer lock implementation, its invariants, and the specifications of its various functions. In Section 5 we conclude.

2 Related Work

We concentrate on related work regarding verification of concurrency primitives. For related work regarding methodology we refer the reader to [4] where the VCC methodology is compared (among others) with the Owicki and Gries method [15], rely / guarantee [11], and concurrent separation logic [14,16] (also in combination with rely / guarantee [18]).

Related work in the area of concurrency primitives deals mostly with specifying primitives and their usage in client code rather than verifying their implementation (such an example is [9] where locks are modeled in an ownership model).

A notable exception of this is [7]. In this work a couple of synchronization primitives including an assembly spin lock implementation for a uniprocessor x86 machine have been verified. All proofs have been conducted in the interactive theorem prover Coq using a logic inspired by separation logic, which has also been proven sound. For the verification of a multiprocessor reader-writer lock implementation extensions to the separation logic (e.g., fractional permissions) and the considered target architecture would be required.

Flanagan et al. [8] show mutual exclusion for a reader-writer lock implementation using a rely / guarantee based prover for multithreaded Java programs. Although they claim that their tools can be applied to large programs they do not present an example. The approach is thread modular (as it is based on rely / guarantee) but not function modular (they simulate function calls by inlining). Compared to our work less annotations are needed to verify the lock. This has several reasons. Their approach and rely / guarantee in general do not properly support data modularity: there is no hiding mechanism, i.e., whenever a single bit in the state is changed *all* guarantees of a thread have to be checked. In the VCC methodology only those invariants are checked that mention the altered data; to make this work additional annotations (e.g., to

describe ownership) are required. Further, their implementation is simplified (a requested write access does not prevent new readers from acquiring shared access) and they do not model dynamic creation / destruction of objects (e.g., initialization or the guarantee that an object still exists during access).

Bornat et al. proposed an ownership scheme to allow for shared read-only state as well [3]. They present a (manual) proof of a reader writer lock implementation based on axiomatically defined semaphores in concurrent separation logic. Besides the missing mechanization, their lock implementation is slightly simpler and less efficient than ours: it is based on semaphores and also may block when releasing a shared lock. Similar to [8] dynamic creation and initialization of locks is not properly modeled.

3 Methodology

The Verifying C Compiler (VCC) [13] is a verifier for concurrent C being developed at Microsoft Research, Redmond, USA, and the European Microsoft Innovation Center (EMIC), Aachen, Germany. VCC is used as a proof tool in the Verisoft XT project [17] to specify and verify industrial software, which includes the Microsoft hypervisor Hyper-V [12]. VCC and the methodology are not purpose-built for our lock example but target a broad class of programs and algorithms.

In this section we give a short overview of the annotation constructs provided by VCC and of the underlying formal model. For the sake of our verification target we pay special attention to concurrency. We illustrate the methodology with small examples. For a more elaborate introduction into the tool and methodology we refer the reader to [13] and, with regards to concurrency, in particular to [4].

Before we start on methodology we give a short introduction on tool work flow. VCC supports adding specifications and other annotations such as hints to the prover directly into the C source code. This includes specification (or ghost) code and objects which do not exist in the real program but are used to support verification. Using conditional compilation, the annotated program can still be regularly compiled. VCC translates the instrumented program into the Boogie language [1]. For the resulting program, the Boogie tool generates verification conditions for partial correctness and passes them to the automatic theorem prover Z3 [6], which ideally succeeds in proving them. Otherwise, a counter example is provided or the prover stops because of running out of resources. In the latter case further diagnostic tools are available for examination. Table 1 gives an overview of some annotations constructs used in this paper. Some of them are explained in more detail below, others

Expressions

set_in(p,S) – set membership
old(e) – refer to pre-state
unchanged(e) – e≡ old(e)

Function contracts

requires(e), **ensures**(E) – pre-, postcondition
writes(s) – function writes to pointers in set s
result – refers to function return value

Objects

invariant(E) – object invariant
wrap(o), **unwrap**(o) – opening and closing objects
owner(o), owns(o) – owner and owns set
span(o) – primitive fields of an object
this – reference to self

Claims

ref_cnt(o) – claim reference count
claim(), **unclaim**() – referencing / dereferencing objects; claim creation / destruction
claims_obj(c,o) – assert target object of a claim
claims(c,e) – assert the property of a claim
stays_unchanged(e) – unchanged during claim's life time

Ghost variables and code

spec() – ghost parameter, variable, or function
claimp() – claim ghost parameter
spec(out x) – by-reference ghost parameter
speconly() – ghost statement

Table 1
VCC Annotation Constructs

are explained when they arise in code.

Objects and Ownership. The current VCC version implements and enforces a Spec#-style object and ownership model [2,9], which has proven to be more elegant and efficient than the low-level memory model supported in earlier versions [5]. In particular, the model guarantees that objects of the same type with different addresses do not overlap in the memory. In the VCC methodology objects may coincide with (pointers to) structures in the annotated C program, but VCC also allows identifying sub structures (so-called groups) and arrays as individual objects. Each object *o* is associated with meta data. Most importantly for this article, *owner(o)* denotes the owner of an object *o*, *owns(o)* denotes the objects it owns, *closed(o)* denotes its closedness, and *ref_cnt(o)* denotes its (claim) reference count. These fields control if and how objects can be accessed, and when object invariants need to or can be assumed to hold. If the owns set of an object is declared as ***vcc****(volatile_owns)* it may change even while the object is closed.

Basically, only open objects can be changed (except for volatile fields) and all objects in the transitive ownership of a closed object are closed as well (more details are given later). In relation to the above mentioned fields Fig. 1 depicts object states and their transition during an object's life time. These can be described as follows:

① After creation, the object is open (i.e., not closed), its reference count is zero, and it is owned by a special object *me()* representing the current thread. Additionally, it is considered *fresh*, i.e., it does not alias with any existing object. In this state, the object *o* is called *mutable*.

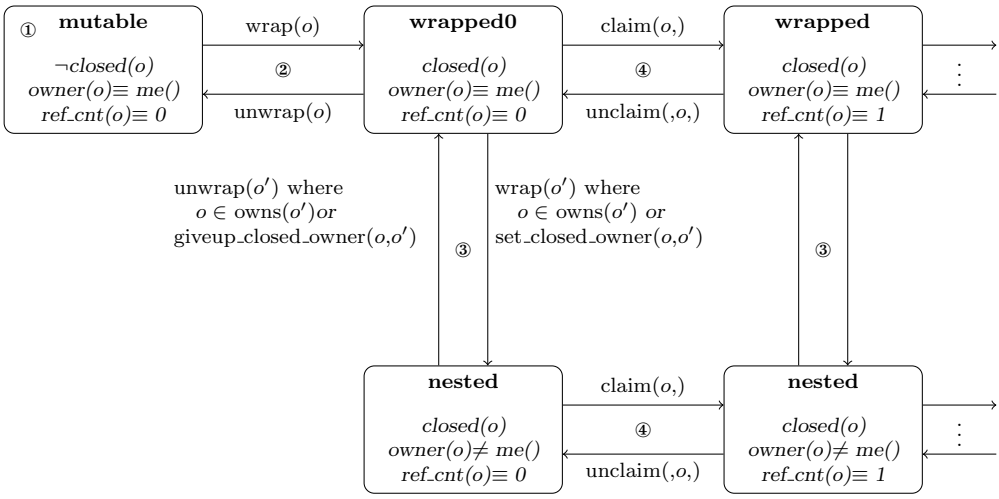

Fig. 1. Objects States, Transitions, and Access Permissions

② By *wrapping* the object (which requires its invariant to hold) it can be closed, while reference count and ownership of the object remain unchanged. The reverse transition is called *unwrapping* and requires the object's reference count to be zero. [4]

③ Closed objects can be added to (and removed from) another object's ownership via the operations ***set_owns()***, ***set_closed_owner()***, and ***giveup_closed_owner()***. The latter two update the ownership set of *closed* objects. If an object containing o in its ownership set becomes (or is already) closed, then o is called *nested*. [5]

④ Lastly, the reference count of an object is modified by ***claim()*** and ***unclaim()*** operations. This gives a way to control when an object can be unwrapped. More details on claims are given later.

Invariants. Objects of aggregate type (e.g., structures) can be annotated with single- or two-state invariants. As a system invariant these invariants are meant to hold for all closed objects (and have to be checked when closing an object): single-state invariants in each state of the system and two-state invariants for each pair of successive states. Two-state invariants are used to restrict possible interference of other threads (this is similar to guarantees in the sense of rely / guarantee reasoning [10,11]).

[4] In [9], the authors use a different terminology although the basic ideas are related. There, wrap and unwrap are called *pack* and *unpack*, respectively. An object which we call mutable is *unshared* there while closed objects would be called *free* or *locked* depending on their owner.

[5] Only for a closed object, its ownership set and the owner fields of the objects it contains are equivalent, which is a rather technical detail.

Note that to reduce the amount of invariant checking, VCC restricts and checks all object invariants to be *admissible*. For details see [4].

Accessing Objects. Access permissions to objects vary according to the object states. A mutable object can be read from or written to. However, write access is only allowed if the object has become mutable within the current function or is listed in the function's ***writes()*** set. Unwrapping an object (which writes the object state) is allowed if the object is listed in the function's ***writes()*** set or has become wrapped in the current function (by unwrapping its parent object). Closed objects that are transitively owned by the current thread are considered thread-local (if no intermediate object has a volatile owns set), and their non-volatile fields can be read. Write accesses to non-volatile fields of closed objects are forbidden.

Volatile accesses to closed objects require a guarantee that the object will not be opened by someone else prior to the access since otherwise the object's invariant cannot be relied upon. There are two ways to obtain such a guarantee: (i) While the object is transitively owned by the current thread, no other thread can open it because thread ownership is a precondition to unwrapping. (ii) While there are *claims* on an object, indicated by a non-zero reference count, it also may not be opened. Thus, a valid claim on an object can be used to justify a volatile access to an object. This is further described below. For a similar reason, read accesses on non-volatile fields of closed objects are permitted by using a claim.

Figure 2 contains a simple example which illustrates some of the concepts introduced so far. [6]

Claims. To capture information on closed, shared objects, VCC provides so-called *claims*. A claim is associated with a number of closed objects (possibly other claims) and guarantees a single-state property on these objects. The claimed property is universally quantified over system states, and can thus be applied in later stages of the verification during the life time of the claim. For soundness reasons only such properties can be claimed that hold initially on claim creation and are stable against interference, i.e., with respect to the two-state invariants of the considered objects.

To allow for any meaningful properties at all, a claim reference counting mechanism is introduced for objects. Every time a claim on an object is created or destroyed, the object's reference count is incremented or decremented, respectively. As a precondition to opening an object its reference count is required to be zero. This will guarantee that claimed objects remain closed

[6] To improve readability we have pretty printed some of the C syntax as mathematical symbols.

```
struct POINT {
  int x, y;
  // Coordinates must be in proper range
  invariant(0 ≤ x ∧ x < 640)
  invariant(0 ≤ y ∧ y < 480)
};

struct RECTANGLE {
  // Upper left and lower right corner
  struct POINT *ul, *lr;
  // We keep (i.e., own) the corners; thus,
     we can rely on their invariants and put
     an additional invariant on their relative
     position
  invariant(keeps(ul, lr))
  invariant(lr→x > ul→x ∧ lr→y > ul→y)
};

void move_x(struct RECTANGLE *r, int
  x)
  writes(r)
  // Require and ensure a wrapped rectangle
  maintains(wrapped(r))
  // Moved rectangle must stay on screen
  requires(r→ul→x + x ≥ 0 ∧ r→lr→x +
    x < 640)
  // Exemplary postconditions: unchanged
     dimensions
  ensures(unchanged(r→lr→x − r→ul→x))
  ensures(unchanged(r→lr→y − r→ul→y))
{
  // Must unwrap rectangle with corners
     before updating coordinates
  unwrap(r);
  unwrap(r→lr); unwrap(r→ul);
  // Can rely on invariants of r, lr, and ul
     here
  r→ul→x += x;
  r→lr→x += x;
  // Wrap the objects in reverse order,
     need to guarantee invariants
  wrap(r→lr); wrap(r→ul);
  wrap(r);
}
```

Fig. 2. Example: Rectangle

for the life time of a claim (and thus their single-state invariant holds), and that it is allowed to assume the claimed objects' two-state invariants in the stability checks of claims.

At the annotation level, claims are represented in VCC as objects of a special (pointer) type ***claim_t***. As a performance optimization, an object not declared as ***vcc****(claimable)* can be assumed to always have a zero reference count. The operation ***claim****$(o_1, \ldots, o_n, p)$* returns a fresh claim referencing objects o_1 to o_n with a claimed property p. VCC checks that write permissions for the referenced objects exists (in order to increment their reference counts), the objects are closed, and the claimed property holds initially and under interference. If these preconditions are met, a valid, fresh claim with the claimed property is returned and the reference counts of the objects are incremented. The operation ***unclaim****$(c, o_1, \ldots, o_n)$* destroys claim c and dereferences object o_1 to o_n. For ***unclaim****()*, write permissions for the referenced objects and claim must exist and, because ***unclaim****()* is a special kind of ***unwrap****()* operation, the claim itself must have a reference count of zero. If these preconditions are met, the claim is opened (nothing meaningful can be done with it anymore) and the reference counts of the objects are decremented.

Atomic Blocks. When objects are closed, VCC allows accesses to volatile fields only inside special atomic blocks, which are meant to represent a *single* system transition. Conceptually, this enables to do thread-modular, sequential

```
atomic (c_1, ..., c_n, o_1, ..., o_m) {
  // Other threads interfere
  // Pre-state for two-state invariant check
  I; // Implementation step
  G; // Ghost step(s)
  // Post-state for two-state invariant check
}
// No further interference
```

```
// Alternative functional form, where the return
// result of I is available in G via the special
// variable 'result'
atomic_op(I, c_1, ..., c_n, o_1, ..., o_m, G)

// Short form to just read from a volatile field
x = atomic_read(f, c_1, ..., c_n, o_1, ..., o_m);
```

Fig. 3. Atomic Block and Alternative Forms

verification outside of atomic blocks, and consider interference of other threads only at the beginning of atomic blocks.

The structure of an atomic block is depicted in Fig. 3 (alongside alternative forms). Each atomic block takes an arbitrary number of valid claims c_1 to c_n and an arbitrary number of (closed) objects o_1 to o_m associated with these claims (mostly we have $n \leq 1$ and $m = 1$). At the beginning of the atomic block, the prover only keeps knowledge about non-shared state, over-approximating the interference of an arbitrary number of other threads and steps. Inside the atomic block, an arbitrary number of ghost statements (including ghost updates) can be executed because scheduling with respect to ghost steps can be assumed benign. Access to implementation variables, however, must be consistent with the atomic operations provided by the underlying architecture (cf. the paragraph **Interlocked Operations** below). The individual statements in the atomic block are checked like regular statements, with extra write permissions to volatile fields of and ghost state owned by the atomic objects o_1 to o_m. Moreover, all statements in the atomic block are combined into a single state transition, which is then checked to be compliant with the two-state invariants of the listed atomic objects. All knowledge required to check the individual statements in the atomic block and the two-state invariants of the objects must be derived from non-shared state, which includes the claimed properties of any non-shared valid claim c (typically one of the listed claims c_1 to c_n).

Interlocked Operations. Most synchronization primitives for concurrent algorithms depend on special atomic operations provided by the underlying architecture; these atomic operations are available in C via compiler intrinsics. For example, the commonly supported compare-exchange operation atomically compares a memory operand against a given value and, if equal, exchanges it with a new value. In addition, the x64 architecture supports a relatively wide range of interlocked bit operations and summation.

During regular compilation the compiler replaces the intrinsics by appropriate assembly instructions. For VCC, we specify the effect of the intrinsics

```
vcc(atomic_inline)
long _InterlockedCompareExchange(
  long volatile *target, long exchange,
    long compare)
{
  long old = *target;
  if (*target == compare) *target =
    exchange;
  return old;
}

vcc(atomic_inline)
long _InterlockedDecrement(long volatile
  *target)
{
  unchecked(*target -= 1);
  return *target;
}
```

```
vcc(atomic_inline)
long _InterlockedOr(long volatile *target,
  long mask)
{
  long old = *target;
  *target |= mask;
  return old;
}

vcc(atomic_inline)
long _InterlockedAnd(long volatile
  *target, long mask)
{
  long old = *target;
  *target &= mask;
  return old;
}
```

Fig. 4. Specification of Interlocked Intrinsics

with atomic inline functions. We give the specification of the intrinsics used in our lock implementation in Fig. 4; the function attribute ***vcc***(*atomic_inline*) makes VCC inline their bodies when used inside ***atomic*** or ***atomic_op***.

Figure 5 illustrates two-state invariants and volatile accesses via claims in a simple example: a countdown with a step function which guarantees (via a claim) that the count will stay zero once this value has been reached. The interlocked compare-exchange operation is used to implement the saturating decrement operation. The ***always***() clause used in the contract of the function is syntactic sugar for requiring and ensuring a wrapped claim with a certain property.

Out Parameters. There is no proper support for out (call-by-reference) parameters in C; instead, one has to pass in a pointer to the object which is to be updated. This extra indirection and the corresponding memory update when the object is updated burden the prover with additional verification conditions. To save some of this effort, VCC supports call-by-reference specification parameters. They are marked with the *out* keyword both in function declarations and calls.

4 Lock Implementation and Verification

In this section we present the implementation of our reader-writer lock and most of its annotations. [7]

[7] Full source with annotations and tests is available at http://www.verisoftxt.de/PublicationPage.html.

```
struct vcc(claimable) COUNTDOWN {
  volatile long i; // Can change
    concurrently
  invariant(i ≥ 0) // Never negative
  // Two-state invariant: do not change or
    decrease by one
  invariant(unchanged(i) ∨ i≡ old(i)−1)
};

long step(struct COUNTDOWN ∗cd
  claimp(c) spec(out claim_t nc))
  writes(c)
  // We do not need ownership of the
    countdown, a claim guaranteeing its
    closedness is enough
  always(c, closed(cd))
  // If non−zero value is returned, nc must
    claim that the countdown has reached
    zero
  ensures(result ⟹ wrapped0(nc) ∧
    claims(nc, closed(cd) ∧ (cd→i ≡ 0)))
{
  long old, new;

  // Atomically read the counter
  old = atomic_read(cd→i, c, cd);
  // Try to decrement
  new = old > 0 ? old − 1 : old;

  // Return success (and a claim) if zero
    was reached (by us or someone else)
  return 0≡ atomic_op(
    _InterlockedCompareExchange(&cd→i,
      new, old),
    c, cd,
    cd→i > 0 ∨ (nc = claim(c, cd→i≡ 0)));
}
```

Fig. 5. Example: Concurrent Countdown

```
#define Write(state)
  ((state)&0x80000000)
#define Readers(state)
  ((state)&0x7FFFFFFF)

typedef struct _LOCK {
  volatile long state;
} LOCK;

void InitializeLock(LOCK ∗lock) {
  lock→state = 0;
}

void AcquireExclusive(LOCK ∗lock) {
  while
    (Write(_InterlockedOr(&lock→state,
    0x80000000))) ;
  while (Readers(lock→state)) ;
}

void ReleaseExclusive(LOCK ∗lock) {
  _InterlockedAnd(&lock→state,
    0x7FFFFFFF);
}

void AcquireShared(LOCK ∗lock) {
  long old_state, new_state;
  do
    {
      do
        old_state = lock→state;
      while (Write(old_state));
      new_state = old_state + 1;
    }
  while (old_state ≠
    _InterlockedCompareExchange(&lock→state,
    new_state, old_state));
}

void ReleaseShared(LOCK ∗lock) {
  _InterlockedDecrement(&lock→state);
}
```

Fig. 6. Reader-Writer Lock Implementation

4.1 Implementation

Figure 6 shows the reader-writer lock implementation. The central data structure *LOCK* contains a single volatile implementation variable called *state*. In its most significant bit the write flag is stored that is set if a client has

```
typedef struct vcc(claimable) vcc(volatile_owns) _LOCK {
  volatile long state;

  spec(volatile obj_t protected_obj;)
  spec(volatile bool writing;)
  spec(volatile claim_t self_claim;)
  spec(volatile bool initialized;)
} LOCK;
```

Fig. 7. Annotated Lock Data Structure

requested exclusive access. The remaining bits hold the number of readers. Putting both values into a single variable is advisable on common architectures in order to enable atomic access. We define the macros *Write()* and *Readers()* to access both values.

The implementation of the lock operations is fairly straightforward. A lock is initialized by setting its *state* variable (and thus the write flag and the number of readers) to zero. Acquiring a lock in exclusive mode proceeds in two phases. First, we spin on setting the write flag of the lock atomically. After the write flag has been set, no new shared locks may be taken. Second, we spin until the number of readers reaches zero. Acquiring a lock in shared mode also proceeds in two phases. First, we spin until the write flag of the lock is reset. This phase is read-only, which prevents cache line thrashing when waiting for the lock to be released in exclusive mode. Second, we try to atomically increment the number of readers. This operation fails if in the meantime someone else took the lock exclusively or shared. In this case we repeat until we succeed in taking the lock. To release a lock in exclusive or shared mode we atomically mask the write flag or decrement the number of readers, respectively.

4.2 Ghost Fields

The annotated declaration of the lock data structure is shown in Fig. 7. In addition to the implementation variable the lock contains four ghost variables. These are: a generic object pointer *protected_obj* identifying the object protected by the lock, a flag *initialized* that is set to one after initialization, a flag *writing* that is one when exclusive access to the protected object has been granted (and all readers have released their locks), and a claim *self_claim*.

The use of *self_claim* is twofold. First, we tie its reference count to the implementation variables of the lock. Unless exclusive access to the lock is granted its reference count will be the sum of the readers count and the write flag. Second, *self_claim* is used as a means to claim properties on the lock. Recall from Section 3 that write permissions on an object are required to

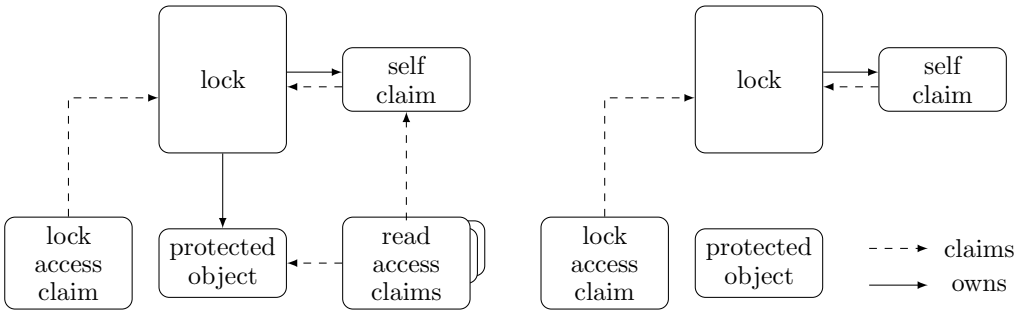

Fig. 8. Ownership and Claims Structure (Shared and Exclusive Access)

establish a claim on it. Usually, once an object (in our case the lock) is closed such write permissions cannot easily be obtained. A self claim serves as a proxy between the claimant and the object. It is a volatile claim owned by the target object and it claims the target object. Thus, in an atomic operation on the target object additional claims on the self claim can be established because of the extra write permissions inside atomic operations. Moreover, these new claims can also establish properties on the target object via the indirection of the self claim.

We make use of this for shared and exclusive lock acquisition. During the acquisition of exclusive locks we need to connect the two loops in the implementation by claiming that after the first loop the write flag has been successfully set. The (ghost) result of shared lock acquisition is a claim stating that the lock will not go into writing mode. In both situations the claimants will know that the self claim cannot become unreferenced. Together with the property on the self claim's reference count the stability of these claims can be established.

Figure 8 depicts the setup of ownership and claims just described. During shared access multiple read access claims may exist. During exclusive access no read access claims exist, and the lock does not own the protected object. The lock access claim shown in the lower left corner of both diagrams is created after initialization of the lock and ensures that the lock remains initialized and allocated. Clients need to pass in their lock access claim (or a claim derived from it) to the lock functions to prove that they are 'allowed' to use the lock.

4.3 Invariants

Figure 9 gives an overview on the dynamic relation between the implementation and specification variables, which we formalize by the invariants on the lock data structure below. Before the *initialized* flag is set, the contents of the other variables do not matter. After initialization, in phases where the write flag is not set, shared locks may be acquired and released, as indicated by the number

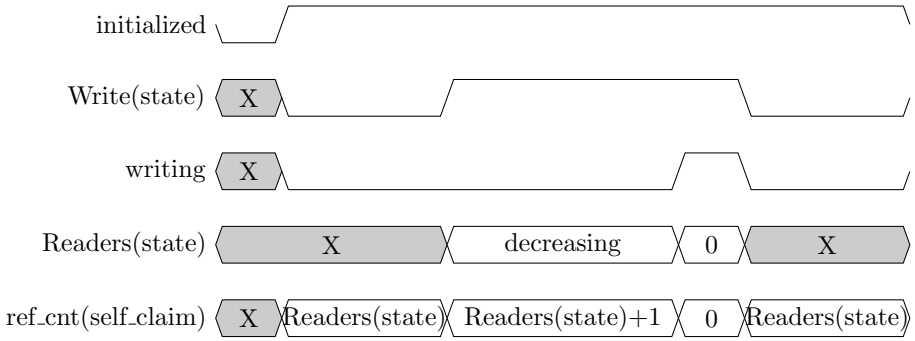

Fig. 9. Relation of Lock Implementation and Specification Variables

of readers. The write flag is set when the acquisition of an exclusive lock starts. In this phase the number of readers must decrease. When it reaches zero, exclusive lock acquisition can complete and the *writing* flag is activated. The self claim's reference count is zero in phases of exclusive access and the sum of the write flag and the number readers else. We now define the invariants formally.

A completed initialization is indicated by the lock being closed and its *initialized* field set to true. In detail, after initialization we require that the protected object is claimable and of aggregate (non-primitive) type, that the self claim is owned by the lock, claims it, and is different from the protected object.

invariant(initialized $\implies$
is_claimable(protected_obj) $\wedge$ is_non_primitive_ptr(protected_obj) $\wedge$
set_in(self_claim, owns(this)) $\wedge$ claims_obj(self_claim, this) $\wedge$
protected_obj $\neq$ self_claim)

Additionally, we need two-state invariants that ensure that the lock stays initialized and that the protected object and the self claim do not change, i.e., are not replaced. For these invariants we need *old(initialized)* instead of *initialized* on the left side of the implication because the two-state invariants on the right side cannot be fulfilled during the activation of *initialized*. [8]

invariant(old(initialized) $\implies$
initialized $\wedge$ unchanged(protected_obj) $\wedge$ unchanged(self_claim))

In the following invariants we assume an initialized lock.

When no exclusive access has been granted (indicated by a zero *writing* flag) the protected object is owned by the lock, its reference count equals the

[8] In two-state invariants *old()* refers to the first of the two states (in contrast to code where it refers to the start state of the current function or loop body).

number of readers, and the reference count of the self claim equals the number of readers plus the write flag (cf. Figs. 8 and 9).

invariant(initialized ∧ ¬writing ⟹
set_in(protected_obj, owns(this)) ∧
ref_cnt(protected_obj) ≡ (**unsigned**) Readers(state) ∧
ref_cnt(self_claim) ≡ (**unsigned**)(Readers(state) + (Write(state)≠ 0)))

When exclusive access to the lock has been requested, the number of readers must decrease and the write flag must not be cleared before the *writing* flag has been set, i.e., an attempt to get exclusive access must not be aborted.

invariant(initialized ∧ old(Write(state)) ⟹
Readers(state) ≤ old(Readers(state)) ∧ (¬Write(state) ⟹
old(writing)))

When exclusive access been granted, the number of readers must be zero, the write flag must be one, and the self claim must be unreferenced.

invariant(initialized ∧ writing ⟹
Readers(state) ≡ 0 ∧ Write(state) ∧ ref_cnt(self_claim) ≡ 0)

4.4 Initialization

From an implementation point of view (as we have seen) initializing a lock simply means clearing its *state* field. From a specification point of view more work has to be done: the lock has to be returned wrapped with its *initialized* and *protected_obj* fields set and a claim needs to be constructed that later allows lock clients to use it.

While we could bundle those specification steps into the implementation function *InitializeLock()*, we have decided to use a second (ghost) function, *ActivateLock()*, to implement these steps. The reason for this separation is that in general the implementation part of lock initialization may precede initialization (and wrapping) of its designated protected object. In such a case, the ghost part of lock initialization must be delayed. Figure 10 shows the annotated versions of both initialization functions.

The pre- and postcondition of *InitializeLock()* are rather straightforward. Its ***writes()*** clause states that the passed-in pointer is mutable and that the content of the lock data structure will be overwritten. The postconditions states that the lock is mutable with its *state* field cleared.

The preconditions of the ghost function *ActivateLock()* include the post-conditions of *InitializeLock()*. The function receives a call-by-reference claim parameter *lock_access_claim*. After lock activation, *lock_access_claim* points to

```
void InitializeLock(LOCK *lock)
  writes(span(lock))
  ensures(mutable(lock))
  ensures(¬lock→state)
{
  lock→state = 0;
}

spec(bool vcc(atomic_inline)
  AcquireObject(LOCK *lock) {
  lock→writing = 0;
  set_closed_owner(lock→protected_obj,
    lock);
  return true;
});

spec(void ActivateLock(LOCK *lock,
  obj_t obj claimp(out lock_access_claim))
  writes(span(lock), obj)
  requires(¬lock→state)
  requires(wrapped0(obj))
  requires(is_claimable(obj))
  ensures(wrapped(lock) ∧ ref_cnt(lock) ≡
    2)
  ensures(wrapped0(lock_access_claim) ∧
    is_fresh(lock_access_claim))
  ensures(claims_obj(lock_access_claim,
    lock))
  ensures(claims(lock_access_claim,
    lock→initialized ∧ lock→protected_obj
      ≡ obj))
{
  lock→initialized = 0;
  set_owns(lock, ∅);
  wrap(lock);
  assert(not_shared(lock));
  atomic (lock) {
    lock→initialized = 1;
    lock→self_claim = claim(lock, true);
    set_closed_owner(lock→self_claim,
      lock);
    lock→protected_obj = obj;
    AcquireObject(lock);
  }
  lock_access_claim = claim(lock,
    lock→initialized ∧ lock→protected_obj
      ≡ obj);
})
```

Fig. 10. Lock Initialization

a fresh (i.e., newly allocated), valid, and unreferenced claim stating that the lock is initialized and its protected object is set to the given object. This claim can be used for lock operations later on.

In the code, the *initialized* flag is cleared and the lock is wrapped (and thereby closed). As we have seen most of the lock invariants are disabled at this point. Only afterwards the lock-internal claims can be set up (since establishing a claim requires the target object to be closed). In detail, we first have to activate the field *initialized*, create the self claim and store the pointer to the protected object in the lock. Second, the *writing* field is also cleared and ownership of the protected lock is taken. Since both steps repeat on exclusive lock release they have been factored out into a separate specification function, *AcquireObject()*.

4.5 Exclusive Access

Figure 11 shows the annotated code of the function *AcquireExclusive()* implementing exclusive lock acquisition. It takes a lock access claim as a ghost parameter. By the ***always()*** clause in the contracts it is required (and ensures) to be valid and guarantee closedness and initialization of the lock. When the function returns it will guarantee to the caller that the protected object is

```
spec(vcc(atomic_inline)
bool CreateWriteFlagClaim(LOCK *lock,
  claimp(out write_flag_claim)) {
  write_flag_claim = claim(lock→self_claim,
    stays_unchanged(lock→self_claim) ∧
    lock→initialized ∧ Write(lock→state));
  return true;
})

spec(vcc(atomic_inline)
bool ReleaseObject(LOCK *lock, claim_t
  write_flag_claim) {
  giveup_closed_owner(lock→protected_obj,
    lock);
  unclaim(write_flag_claim,
    lock→self_claim);
  lock→writing = 1;
  return true;
});

void AcquireExclusive(LOCK *lock
  claimp(lock_access_claim))
  always(lock_access_claim, closed(lock) ∧
    lock→initialized)
  ensures(wrapped0(lock→protected_obj)
    ∧ is_fresh(lock→protected_obj))
{
  spec(claim_t write_flag_claim;)

  while
    (Write(atomic_op(_InterlockedOr(&lock→state,
    0x80000000),
    lock, lock_access_claim,
    Write(result) ∨
      CreateWriteFlagClaim(lock spec(out
      write_flag_claim)))))
    ;

  while (Readers(atomic_op(lock→state,
    lock, write_flag_claim,
    Readers(result) ∨ ReleaseObject(lock,
      write_flag_claim))))
    invariant(wrapped0(write_flag_claim))
    ;
}
```

Fig. 11. Acquisition of an Exclusive Lock

unreferenced, wrapped, and fresh (and thus, writable).

In the first loop of the implementation we spin until the write flag could be atomically set (via the *_InterlockedOr* intrinsic), i.e., in an atomic block the write has been seen as zero and then set to one. The last parameter to the atomic operation has the effect that a temporary claim *write_flag_claim* is generated via the atomic inline function *CreateWriteFlagClaim()* if the write bit of the return value of *_InterlockedOr* (i.e., the value of *lock→state* before the interlocked update) is zero.[9] This temporary claim references the self claim and states that the lock stays initialized, that the self claim stays, and that the write flag of the lock has been set. The claimed property holds initially by virtue of the passed-in lock access claim and the update of the interlocked operation. The property is also stable because as long as there remains a reference to the self claim, the *writing* flag cannot be activated and the write flag cannot be reset (cf. Section 4.3). This also guarantees unchangedness of the self claim. The atomic update satisfies the lock invariant.

The second loop waits for the readers to disappear. If the number of readers has been seen as zero, we remove the protected object from the ownership of the lock via the atomic inline function *ReleaseOwner()*, which discards

[9] Since VCC expects an expression as last parameter of the atomic operation we make use of lazy evaluation.

```
void ReleaseExclusive(LOCK *lock
  claimp(lock_access_claim))
  always(lock_access_claim,
    closed(lock) ∧ lock→initialized ∧
      stays_unchanged(lock→protected_obj))
  requires(lock_access_claim ≠
    lock→protected_obj)
  requires(wrapped0(lock→protected_obj))
  writes(lock→protected_obj)
{
  atomic (lock, lock_access_claim) {
    _InterlockedAnd(&lock→state,
      0x7FFFFFFF);
    speconly(AcquireObject(lock);)
  }
}
```

Fig. 12. Release of an Exclusive Lock

```
void AcquireShared(LOCK *lock
  claimp(lock_access_claim) claimp(out
    read_access_claim))
  always(lock_access_claim, closed(lock) ∧
    lock→initialized)
  ensures(wrapped0(read_access_claim) ∧
    is_fresh(read_access_claim))
  ensures(claims(read_access_claim,
    closed(lock) ∧
    lock→initialized ∧ ¬lock→writing ∧
    claims_obj(read_access_claim,
      lock→self_claim) ∧
    claims_obj(read_access_claim,
      lock→protected_obj) ∧
    closed(lock→protected_obj)))
```

```
void ReleaseShared(LOCK *lock
  claimp(read_access_claim))
  writes(read_access_claim)
  requires(wrapped0(read_access_claim))
  requires(claims(read_access_claim,
    closed(lock) ∧
    lock→initialized ∧ ¬lock→writing ∧
    claims_obj(read_access_claim,
      lock→self_claim) ∧
    claims_obj(read_access_claim,
      lock→protected_obj)))
```

Fig. 13. Specifications of Acquire and Release Shared Lock

the temporary claim and sets the writing field to one. All of this can be justified by the claimed property of *write_flag_claim* and the lock's invariant. Setting the writing field is allowed because the write flag is known to be active. Furthermore, the writing flag is known to be zero in the pre-state of the atomic operation because the reference count of the self claim, which is referenced by *write_flag_claim*, cannot be zero. This justifies the remaining operations.

Releasing an exclusive lock (cf. Fig. 12) is simpler. The operation requires the protected object to be wrapped and unreferenced. Furthermore, a claim is required that guarantees the closedness and initialization of the lock as well as the value of its protected object. In the implementation, the write flag of the lock is atomically masked out via an *_InterlockedAnd()* intrinsic. In the same machine step, the atomic inline function *AcquireOwner()*, which we have already seen in Section 4.4, takes back ownership of the protected object (including accompanying ghost operations).

4.6 Shared Access

Figure 13 shows the specifications of the lock functions for shared access; due to space restrictions the annotated code is not presented. The function

AcquireShared() passes out a claim parameter *read_access_claim*. After returning, *read_access_claim* needs to be a fresh, valid, wrapped, and unreferenced claim, which derives from the self claim and the protected object. The latter fact allows using the returned claim in read accesses (and volatile write accesses) on the protected object. The function *ReleaseShared()* takes back such an unreferenced claim and destroys it.

5 Conclusion

We have presented the formal verification of a realistic multi-processor reader-writer lock implementation in C. For exclusive access, callers get full ownership of the protected object. For shared access, callers get read permissions and a guarantee that the protected object is not changed or destroyed. Specifications and proofs are modular, as encouraged by the VCC methodology. The effort to verify the reader-writer lock implementation with six functions and four additional test functions (in total about 250 lines of code and annotations) was approximately four person weeks – including some getting used to VCC and the verification methodology. VCC checks the implementation in 4 seconds on one core of a 2.2GHz Intel Core 2 Duo machine. We have also successfully applied the methodology during verification of the more complex reader-writer lock implementation of Microsoft's hypervisor Hyper-V and used the resulting specifications successfully in the verification of client code.

We have several ideas how to extend our results in future work. We want to deal with lock destruction, which should unwrap the lock and return the protected object in a wrapped state. This would require extensions of the invariants, because currently clients are not forced to give up the lock before returning their lock access claim. These extensions might also be used to prevent common errors of lock usage leading to deadlocks. Also, our work should be extended to cover other concurrency primitives and their implementation as well.

Acknowledgement

We wish to thank Ernie Cohen, Michał Moskal, and Stephan Tobies for their feedback on earlier versions of this paper and helpful discussions.

References

[1] Barnett, M., 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, *FMCO 2005*, LNCS **4111** (2006), pp. 364–387.

[2] Barnett, M., K. R. M. Leino and W. Schulte, *The Spec# programming system: An overview*, in: *CASSIS*, LNCS **3362** (2004), pp. 49–69.

[3] Bornat, R., C. Calcagno, P. W. O'Hearn and M. J. Parkinson, *Permission accounting in separation logic*, in: J. Palsberg and M. Abadi, editors, *POPL 2005* (2005), pp. 259–270.

[4] Cohen, E., M. Moskal, W. Schulte and S. Tobies, *A practical verification methodology for concurrent programs*, Technical Report MSR-TR-2009-15, Microsoft Research (2009).

[5] Cohen, E., M. Moskal, W. Schulte and S. Tobies, *A precise yet efficient memory model for C*, in: *4th International Workshop on Systems Software Verification (SSV 2009)*, Electronic Notes in Theoretical Computer Science (2009).

[6] de Moura, L. and N. Bjørner, *Z3: An efficient SMT solver*, in: C. R. Ramakrishnan and J. Rehof, editors, *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, Lecture Notes in Computer Intelligence **4963** (2008), pp. 337–340.

[7] Feng, X., Z. Shao, Y. Dong and Y. Guo, *Certifying low-level programs with hardware interrupts and preemptive threads*, in: R. Gupta and S. P. Amarasinghe, editors, *PLDI 2008* (2008), pp. 170–182.

[8] Flanagan, C., S. N. Freund and S. Qadeer, *Thread-modular verification for shared-memory programs*, in: *ESOP '02: Proceedings of the 11th European Symposium on Programming Languages and Systems* (2002), pp. 262–277.

[9] Jacobs, B., J. Smans, F. Piessens and W. Schulte, *A simple sequential reasoning approach for sound modular verification of mainstream multithreaded programs*, Electronic Notes in Theoretical Computer Science **174** (2007), pp. 23–47.

[10] Jones, C., "Development Methods for Computer Programs Including a Notion of Interference," Ph.D. thesis, Oxford University (1981).

[11] Jones, C. B., *Tentative steps toward a development method for interfering programs*, ACM Transactions on Programming Languages and Systems **5** (1983), pp. 596–619.

[12] Microsoft Corp., *Verisoft – Formal verification of computer systems*, http://www.microsoft.com/emic/verisoft.mspx (2007).

[13] Microsoft Corp., *VCC: A C Verifier*, http://research.microsoft.com/en-us/projects/vcc/ (2008).

[14] O'Hearn, P. W., *Resources, concurrency, and local reasoning*, Theoretical Computer Science **375** (2007), pp. 271–307.

[15] Owicki, S. and D. Gries, *Verifying properties of parallel programs: An axiomatic approach*, Communications of the ACM **19** (1976), pp. 279–285.

[16] Reynolds, J. C., *Separation logic: A logic for shared mutable data structures*, 2002, pp. 55–74.

[17] The Verisoft XT Consortium, *The Verisoft XT Project*, http://www.verisoftxt.de/ (2007).

[18] Vafeiadis, V. and M. J. Parkinson, *A marriage of rely/guarantee and separation logic*, in: L. Caires and V. T. Vasconcelos, editors, *CONCUR*, Lecture Notes in Computer Science **4703** (2007), pp. 256–271.