Verification of Shared-Reading Synchronisers

Afshin Amighi
Hogeschool Rotterdam*
Rotterdam, The Netherlands
a.amighi@hr.nl

Marieke Huisman
University of Twente
Enschede, The Netherlands
m.huisman@utwente.nl

Stefan Blom
BetterBe*
Enschede, The Netherlands
sblom@betterbe.com

Synchronisation classes are an important building block for shared memory concurrent programs. Thus to reason about such programs, it is important to be able to verify the implementation of these synchronisation classes, considering atomic operations as the synchronisation primitives on which the implementations are built. For synchronisation classes controlling exclusive access to a shared resource, such as locks, a technique has been proposed to reason about their behaviour. This paper proposes a technique to verify implementations of both *exclusive access* and *shared-reading* synchronisers. We use permission-based Separation Logic to describe the behaviour of the main atomic operations, and the basis for our technique is formed by a specification for class AtomicInteger, which is commonly used to implement synchronisation classes in java.util.concurrent. To demonstrate the applicability of our approach, we mechanically verify the implementation of various synchronisation classes like Semaphore, CountDownLatch and Lock.

1 Introduction

As our society is increasingly becoming more digital, there is an urgent need for techniques that can improve the performance of software. Concurrency is commonly used to achieve this goal, as it allows to split bigger tasks into multiple smaller tasks, which can be executed simultaneously. This has many advantages, but also a major disadvantage, namely that developing concurrent software is error-prone, as one has to keep track of how all threads can interact with each other. Therefore, we need techniques that allow to specify and verify the behaviour of concurrent programs.

In this paper, we consider this problem. We focus in particular on shared memory concurrent programs, where multiple threads interact and communicate via a common, shared memory. One of the main building blocks of such programs are synchronisation classes that control the access to a shared memory. We distinguish between two important classes of synchronisers: *exclusive accesses* synchronisers, such as locks, that ensure that always at most one thread at a time can access a shared memory location, and *shared-reading* synchronisers, such as semaphores, that also allow multiple threads to read the same shared location simultaneously. The concurrency API java.util.concurrent (JUC) provides several variations of both kinds of synchronisers, typically implemented on top of the AtomicInteger class.

We proposed a technique to specify and verify *exclusive access* synchronisers, such as Lock, using permission-based Separation Logic [3]. Our paper extends this approach to cover also *shared-reading* synchronisers. The original approach identifies two main components that make up the specification of a synchroniser: (1) the value of the atomic variable, i.e. the atomic state, and (2) the views of the participating threads on the atomic state, i.e. the latest value that a thread remembers from the atomic state. The client program using the synchroniser specifies a synchronisation protocol that captures the roles of the threads and a resource invariant describing the shared memory location protected by the synchroniser.

*The research is done while working at University of Twente.

Submitted to:
MeTRiD 2018

© A. Amighi, M. Huisman & S. Blom
This work is licensed under the
Creative Commons Attribution License.

In this paper, we make this approach more fine-grained, allowing a thread to obtain only a read permission to the shared memory location. We derive new specifications for the atomic operations that capture the possibility of obtaining both exclusive and partial access, and combine these into a new contract for the class AtomicInteger.

Applicability of the approach is demonstrated by discussing the verification of several commonly used synchronisers: Semaphore, CountDownLatch and Lock. All examples are mechanically verified using our VerCors tool-set [4].

The paper is structured as follows: Section 2 briefly introduces permission-based Separation Logic. Section 3 discusses several implementations of typical shared-reading synchronisers. Section 4 derives the specifications for the three main atomic operations, using permission-based Separation Logic. Then, Section 5 combines this into a contract for AtomicInteger, and Section 6 shows how this is used to verify the synchronisation classes. Finally, Section 7 concludes the paper, and discusses related work.

2 Background

This section briefly explains permission-based Separation Logic (PBSL) [5] and its role in reasoning about concurrent programs. Concurrent Separation Logic (CSL) [18], which is an extension of SL [21], is a Hoare-style program logic to reason about *multi-threaded* programs. In addition to the predicates and operators from first order logic, CSL uses two new constructs in the specifications: (1) The *points-to* predicate: $e \mapsto v$ describes that the location of the heap addressed by e is pointing to a location that contains the value v, and (2) The $*$*-conjunction* operator: $\phi * \psi$ expresses that predicates ϕ and ψ hold for two disjoint parts of the heap. We use $[e]$ to denote the contents of the heap at location e, and we use $e \mapsto -$ to indicate that the precise contents stored at location $[e]$ is not important. Predicates P and Q of a Hoare-triple $\{P\}\ C\ \{Q\}$ in CSL are predicates on the state where the state is a pair of the store and the heap. The key point of verification using CSL is the *ownership* concept. In the verification of C if its precondition P asserts $e \mapsto v$, it is assumed that the executing thread t has the full ownership of e. This means that no other thread can interfere with t to update e, unless t transfers the ownership of e to another program.

O'Hearn developed required rules to reason about threads exchanging *exclusive* ownership of a memory location through a synchronisation construct [18]. In the rules related to shared memory, the shared state is specified by a *resource invariant*: a predicate that expresses the properties of the shared variables that must be preserved in all the states visible by all the participating threads. The general judgement in CSL, denoted as $I \vdash \{P\}\ C\ \{Q\}$, expresses that with a resource invariant I, the execution of C satisfies the Hoare triple $\{P\}\ C\ \{Q\}$. The resource invariant can be obtained by executing operations of the associated synchroniser. For example, a concurrent program synchronised with a single-entrant lock can be verified as follows [11]: any thread that successfully obtains the lock acquires I and before releasing the lock it has to detach I from its local state. Verification of an atomic operation proceeds similar to verification of a class using a lock: (1) the thread executing an atomic operation acquires the global lock, (2) it adds the shared resources that are captured by the resource invariant to its local state, (3) it performs its action on the resources, (4) it establishes the resource invariant, and finally, (5) it releases the lock by separating itself from the resource invariant (and all this is done atomically). This is described formally by the following rule of Vafeiadis [25]:

$$\frac{\mathsf{emp} \vdash \{P * I\}\ C\ \{I * Q\}}{I \vdash \{P\}\ \mathsf{atomic}\{\ C\ \}\ \{Q\}} \quad [\textsc{Atomic}]$$

where I is the resource invariant, emp is the empty heap, atomic$\{\ C\ \}$ indicates that the command C is executed atomically, P is the precondition for execution of the atomic operation, and Q is the postcondition of the atomic operation.

To enable reasoning about multiple threads simultaneously reading the same shared data, CSL has been extended with permissions [6] to PBSL. This extension is necessary to specify and verify *shared-reading* synchronisations [5]. In PBSL, any access to location of the heap is decorated with a fractional permission $\pi \in (0,1]$. Any fraction $\pi \in (0,1)$ is interpreted as a *read* permission and the full permission $\pi = 1$ denotes a *write* permission (full ownership). Permissions can be transferred between threads at synchronisation points (including thread creation and joining). A thread can only mutate a location if it has the write permission to that location. Based on the following rule, permissions can be split and combined to change between read and write permissions: $e \overset{\pi}{\mapsto} v * e \overset{\pi'}{\mapsto} v \Leftrightarrow e \overset{\pi+\pi'}{\mapsto} v$, where $\pi + \pi'$ is undefined if the result is greater than 1.

Soundness of the logic ensures that the sum of all permissions to a location is never more than 1. Thus, at most one thread at a time can be writing to a location, and whenever a thread has a read permission, all other threads holding a permission on this location simultaneously must have a read permission. As a result, a verified concurrent program using PBSL is data-race free. This makes PBSL to specify the behaviour of a shared-reading synchronisation mechanism.

In this paper we are using our VERCORS specification language to specify and verify the behaviour of synchronisers (in Section 5). The specification language of VERCORS is an extension of the Java Modeling Language (JML) with PBSL. The standard SL notation of $*$ for separating conjunction becomes $**$ in our specifications, in order to avoid a syntactical clash with the multiplication operator of Java (and JML). Method and class specifications can be preceded by a **given** clause, declaring ghost parameters to method and classes. Ghost method parameters are passed at method calls, ghost class parameters are passed at type declaration and instance creation, resembling the parametric types mechanism of Java. This mechanism is used to pass resource invariants to classes. Furthermore, the language has support to declare abstract predicates [20], by providing the name, typing and parameter declaration.

The full grammar for the VERCORS specification language is as follows:

$$
\begin{array}{lcl}
R & ::= & B \mid \mathsf{Perm}(\mathsf{field},\ \mathsf{pi}) \mid (\backslash\textbf{forall}*\ \mathsf{T}\ \mathsf{i}; B; R) \\
 & \mid & R_1 ** R_2 \mid B_1 ==> B_2 \mid E.\mathsf{P}(E_1, \cdots, E_2) \\
E & ::= & \text{any } \textit{pure} \text{ expression} \\
B & ::= & \text{any } \textit{pure} \text{ expression of type boolean}
\end{array}
$$

where R denotes *resource expressions* (typical elements r_i), E represents *functional expressions* (typical elements e_i), B is the logical expressions of type boolean (typical elements b_i), T is an arbitrary type, vi is a variable name, P is an abstract predicate of a special type **resource**, field is a field reference, and pi denotes a fractional permission.

3 Shared-reading Synchronisers

In Java, volatile variables can be used as a communication mechanism between multiple threads. Writing to (or reading from) a volatile field has the same memory effect as if a monitor is released (or locked). Therefore when writing to a volatile variable, its value becomes immediately visible to other threads. This guarantees that reading a volatile field, always gets the latest completed written value. This is an essential feature for synchronisers, because all threads must have a *consistent view* on the state of the synchroniser.

```
public class Semaphore{
  private AtomicInteger sync;
  Semaphore(int n){
    sync = new AtomicInteger(n); }

  public void acquire(){
    boolean stop = false; int c = 0;
    while(!stop) {
      c = sync.get();
      if( c > 0 ){
        int nextc = c−1;
        stop = sync.compareAndSet(c,nextc);
      }
    }
  }
  public void release(){
    boolean stop = false;
    while(!stop) {
      int c = sync.get();
      int nextc = c+1;
      stop = sync.compareAndSet(c,nextc);
    }
  }
}
```

Listing 1: Semaphore: Implementation.

```
public class CountDownLatch{
  private AtomicInteger sync;
  CountDownLatch(int count){
    sync=new AtomicInteger(count); }

  void countDown(){
    boolean stop = false;
    int c = 0 , nextc = 0;
    while(!stop){
      c = sync.get();
      if (c > 0){
        nextc = c−1;
        stop = sync.compareAndSet(c, nextc);
      }
    }
  }
  void await(){
    int c = sync.get();
    while(c!=0) {  c = sync.get(); }
  }
}
```

Listing 2: CountDownLatch: Implementation.

The atomic package of JUC contains a set of atomic classes that define wrapper functions for private volatile fields with different types. Each atomic class defines three basic atomic operations. For example the AtomicInteger class exports get() for atomic read, set(**int** v) for atomic write and compareAndSet(**int** x,**int** n) for atomic conditional update. The compareAndSet(**int** x,**int** n) method first atomically checks the current value of the volatile field and updates it to n if it is equal to the expected value x, otherwise it leaves the state unchanged, and then it returns a boolean to indicate whether the update succeeded. This AtomicInteger class is the basis for almost all synchroniser implementations in java.util.concurrent, such as ReentrantLock and other classes implementing the interface Lock, Semaphore, CyclicBarrier and CountDownLatch.

Here we present (simplified) implementations of two different shared-reading synchronisation classes: Semaphore and CountDownLatch. In our implementations, we stripped fairness conditions from the original soure code, *i.e.* we did not implement any algorithm to fairly pick the next candidate for the shared resource competition. These examples illustrate how atomic variables are used in shared-reading synchronisers, which will help us to explain the formal specification in Section 4. Finally, in Section 6 we will demonstrate how these synchronisers are verified.

In a Semaphore (see Listing 1) all participating threads compete with each other to acquire or release protected portions of the shared resource. In a concurrent program synchronised with a semaphore, any thread trying to acquire a portion, has to win the competition by atomically decrementing the number of available portions (see line 12 of Listing 1). Similarly, as implemented in line 22 a releasing thread (again in a competition) must atomically increment the number of available portions.

Next we consider a CountDownLatch. Suppose we have an application with disjoint sets of active

and passive threads, where active threads initially own a portion of the shared resource and passive threads wait for active threads to release their portions. CountDownLatch, as implemented in Listing 2 blocks all the passive threads, until all active threads have released their portion of the shared resource. If the passive threads are unblocked, ownership of the shared resource is transferred to the passive threads.

A CountDownLatch maintaints a counter that denotes the number of active threads working on the shared resource. Each active thread, once finished, calls countDown() on the latch, which decreases the counter (see line 10), to signal that it is done. The passive threads wait for the active threads by calling the blocking await() method on the latch. Inside this method, the passive threads are continuously reading the state of the latch until it reaches zero (line 20). In fact, the latch collectively accumulates the full shared resource from the active threads and the waiting passive threads can continue their task only when they see that there is no more active thread possessing a portion of the shared resource.

In summary, groups of threads involved in the synchronisation can be abstracted by their behavioral *role*. If threads with an identical role share a resource (as in Semaphore), then in order to obtain (or release) a portion of that resource, they have to participate in a compareAndSet-based competition. But, if threads have different roles (as the passive and active thread groups in CountDownLatch) they can exchange the shared resource by reading the atomic variable that controls the access. Note however, active threads in the CountDownLatch still have to compete with each other to release their portions. In both of these synchronisers, the state of the volatile counter defines the remaining portion of the shared resource. Intuitively, associating the role of the threads, the state of the synchroniser and the portions of the shared resources distributed among the threads are the main elements in our reasoning about synchronisers which is explained in the next section.

4 Reasoning about Atomics

This section extends the formal specifications of the atomic operations presented in [3] in such a way that they can be used to verify both exclusive access and shared-reading synchronisation constructs.

We [3] identified various synchronisation patterns using basic atomic operations. These synchronisation patterns show that a thread: (1) can both obtain or release resources by calling the compareAndSet (or simply `cas`) operation, if it wins the competition, (2) may only obtain resources by calling the `get` operation, provided it meets the conditions imposed by the protocol on the thread's view of the atomic variable, and (3) always releases resources by writing an atomic location using the `set` operation.

To explain the essence of our specification, first, we focus on competitive resource acquisition using the `cas` operation. We start with a simple example that illustrates the behavior of atomic variables to see how a fraction of the shared resource is exchanged when an atomic variable is used as a shared-reading synchronisation mechanism.

Similar to the formalisation for exclusive access synchronisers [3], we partition the heap augmented with permissions into two disjoint parts, denoted ALoc for atomic locations and NLoc for non-atomic locations. For a given atomic variable $s \in \mathsf{ALoc}$, we restrict the set of atomic operations to: (1) $\texttt{get}(s)$ for atomic read of s, (2) $\texttt{set}(s,n)$ for atomic update of s with the value n, and (3) $\texttt{cas}(s,x,n)$ for conditional atomic update of s from the value x to the value n. Resources are defined as locations from the non-atomic part of the heap. Further, we extend the interval of the permissions to include 0 and we define $e \overset{0}{\mapsto} - \equiv \mathsf{emp}$ (emp denotes the empty heap).

As an example, using a semaphore $s \in \mathsf{ALoc}$ to protect a location $r \in \mathsf{NLoc}$, the value of the atomic location s (defined as atomic state) indicates the number of available fractions for the semaphore. The resource invariant for s associates the value of s with the maximum number of threads that concurrently

can read r and is defined as:

$$I_s = \exists\, v \in \{0,\cdots,M\}.\ s \overset{1}{\mapsto} v * r \overset{v}{\underset{M}{\mapsto}} -$$

In an implementation of the semaphore, any thread that wishes to acquire a portion of the shared resource must atomically decrement the value of s by 1. This transfers $\frac{1}{M}$ of r from s to the calling thread. This fraction is stored back to s by releasing the semaphore, which increments the current value of s atomically by 1.

In the implementations of acquire and release, the executing thread with expected value x executes the atomic body of the cas operation. As justified by the atomic rule, to verify the body it obtains I_s. This gives full access of s, as well as $\frac{x}{M}$ of r (provided the current state equals x). The thread then updates s to $n = x-1$ for acquire or $n = x+1$ for release, and re-establishes I_s with $r \overset{n}{\underset{M}{\mapsto}} -$ before leaving the body. To do so, the thread either acquires itself a $(\frac{x}{M} - \frac{n}{M})$ fraction of r or it releases a $(\frac{n}{M} - \frac{x}{M})$ fraction of r. This example gives us the necessary intuition to derive a specification for the cas operation to cover both *shared and exclusive* synchronisers.

If we denote the shared resources to be protected by the atomic location s using $\mathsf{res}(s)$, then we can define the resource invariant as:

$$I_s = \exists\, v \in \{0,\cdots,M\}.\ s \overset{1}{\mapsto} v * S(s,\pi)$$

where $S(s,\pi) = \circledast_{r \in \mathsf{res}(s)}\ r \overset{\pi}{\mapsto} -$.

Using I_s, the atomic location s is interpreted as the owner of the resources for which the threads compete through the cas operation in order to obtain or release their permissions. Based on this general definition of resource invariant, we can specify the behavior of cas. For a synchroniser s, if p maps the state of the synchroniser to the fractions with a maximum number of threads M, then we can axiomatise cas as follows:

$$\frac{\pi = p(s,x,M) \quad \pi' = p(s,n,M) \quad b = \mathtt{cas}_\tau(s,x,n)}{I_s \vdash \begin{array}{l} \{S(s,\pi' \dot{-} \pi)\} \\ \mathtt{cas}_\tau(s,x,n) \\ \{(b \implies S(s,\pi \dot{-} \pi')) * (\neg b \implies S(s,\pi' \dot{-} \pi))\} \end{array}} \ [\textsc{CAtm}]$$

where $\dot{-}$ denotes the cut-off subtraction over the fractions in $[0,1]$, defined as follows:

$$\pi \dot{-} \pi' = \begin{cases} \pi - \pi' & \text{iff } \pi \geq \pi', \\ 0 & \text{otherwise} \end{cases}$$

Surprisingly, the behaviors of both atomic read and write are more subtle than for the cas operation. This is because their behavior can differ from one case to another. In some cases, the atomic read operation only updates the knowledge of the executing thread without transferring any resources: see lines 9 and 20 in Listing 1, line 10 in Listing 2. Also, the waiting threads in CountDownLatch (see line 20 from Listing 2) obtain their fractions only when they realize that the latch has reached zero. In other cases, unconditional updates in the atomic writes require a *rely-guarantee* [13] style of reasoning as the writing thread must adhere to a *protocol* which guarantees the safety of the write to the environment [3]. This is thoroughly discussed and formalised by Amighi *et al.*[3]. Here we extend the formal definition of the resource invariant from [3] to associate the state of the atomic variable with the fractions of the resources. First, we explain some notations which are used in the definitions.

A thread view is an *atomic ghost field* defined for each thread that stores the last visited value of the atomic state. Each view is indexed by the owning thread identifier and the ownership of a view is split

between the owning thread and the resource invariant, thus, it can only be updated inside an atomic body. $\vec{s_t}$ denotes the vector of views, indexed by their thread identifiers. A vector of values pointed to by the views, indexed by the corresponding thread identifiers, is written $\vec{v_t}$, while $\vec{v_{t\{v_\tau = x\}}}$ denotes a vector such that the value of the item indexed with τ in the vector of values $\vec{v_t}$ is equal to x. Finally, having defined the views of the threads the synchronisation construct is generalised from a single atomic location s to a tuple of the atomic location plus all the thread views of this location.

We decomposed the resource invariant into two components [3]. The first component is the global resource invariant that associates the resources to the *global* atomic state:

$$I_s = \exists v, \vec{w_t} \in \mathsf{Val} \cdot s \overset{1}{\mapsto} v * (\circledast_{t \in \mathsf{Thr}} s_t \overset{\frac{1}{2}}{\mapsto} w_t) * S(s,\pi) * \mathsf{P}_s^{\mathsf{Thr}}$$

where:

- having fsbl for determining the feasibility of the values taken by the atomic location and all the thread views $\mathsf{P}_s^{\mathsf{Thr}}$ is defined as follows:

$$\mathsf{P}_s^{\mathsf{Thr}} = \bigvee_{v, \vec{w_t} \in \mathsf{Val} \cdot \mathsf{fsbl}(v, \vec{w_t})} ([s] = v \wedge [\vec{s_t}] = \vec{w_t})$$

- the fraction of the resources is associated with the atomic state via $\pi = p(s, v, M)$, and
- finally, $S(s,\pi) = \circledast_{r \in \mathsf{res}(s)} r \overset{\pi}{\mapsto} -$

The second component associates the fractions of the resources to the thread views which can be exchanged through a collaborative synchronisation:

$$T(s_t, \pi) = \circledast_{r \in \mathsf{res}(s_t)} r \overset{\pi}{\mapsto} - \text{ where } \pi = p(s_t, w_t, M)$$

By giving a definition for $T(s_t, \pi)$ one can express when a thread with a particular knowledge may obtain the resource. The `set` absorbs the resources either through I_s to the atomic location or through $T(s_t, \pi)$ to the reader thread. This is formally specified in the contracts for the basic atomic operations which are presented in Figure 1.

Comparing the new contribution with [3], our formalised extension for shared-reading synchronisers can be summarised by the following steps: (1) an extension of the permission interval with 0, (2) associating the fractions of the shared resource to the global atomic state, (3) and updating the contract of `cas` using the cut-off subtraction operation.

The next section presents how the specification from Figure 1 translates into a contract for the AtomicInteger class, using our VERCORS [27] specification language.

5 Contract of Atomic Integer

The new contract of AtomicInteger is presented in Listing 3. First we summarise the elements of the contract that are defined from our earlier work [3]. Then, we explain our extensions regarding shared-reading synchronisers.

Listing 3 shows how the AtomicInteger class is parametrised by rs, inv, share, trans, where rs is a set of roles abstracting participating threads, inv represents an abstract predicate as a resource invariant, specifying the shared resources to be protected by AtomicInteger, and share defines a function to associate

$$\frac{\pi = p(s,n,M) \quad \pi' = p(s_\tau,\mathsf{d},M) \qquad \forall\, v, \vec{v_t} \in \mathsf{Val}.\ v_\tau = \mathsf{d} \wedge \mathsf{fsbl}(v, \vec{v_t}_{\{v_\tau=\mathsf{d}\}}) \implies \mathsf{fsbl}(n, \vec{v_t}_{\{v_\tau=n\}})}{I_s \vdash \quad \{s_\tau \overset{\frac{1}{2}}{\mapsto} \mathsf{d} * \mathsf{S}(s,\pi) * \mathsf{T}(s_\tau,\pi')\}\ \mathtt{set}_\tau(s,n)\ \{s_\tau \overset{\frac{1}{2}}{\mapsto} n\}} \quad [\textsc{WAtm}]$$

$$\frac{\pi = p(s_\tau,M) \quad \pi' = p(s_\tau,M)}{I_s \vdash \quad \{s_\tau \overset{\frac{1}{2}}{\mapsto} \mathsf{d} * \mathsf{T}(s_\tau,\pi)\}\ \mathtt{get}_\tau(s)\ \{s_\tau \overset{\frac{1}{2}}{\mapsto} \mathit{ret} * \mathsf{T}(s_\tau,\pi')\}} \quad [\textsc{RAtm}]$$

$$\frac{\begin{array}{l}\pi = p(s,x,M) \quad \pi' = p(s,n,M) \\ \forall\, v, \vec{v_t} \in \mathsf{Val}.\ v_\tau = x \wedge \mathsf{fsbl}(v, \vec{v_t}_{\{v_\tau=x\}}) \implies \mathsf{fsbl}(n, \vec{v_t}_{\{v_\tau=n\}}) \quad b = \mathtt{cas}_\tau(s,x,n)\end{array}}{\begin{array}{ll} & \{s_\tau \overset{\frac{1}{2}}{\mapsto} x * \mathsf{S}(s,\pi' \dot{-} \pi)\} \\ I_s \quad \vdash & \mathtt{cas}_\tau(s,x,n) \\ & \{(b \implies s_\tau \overset{\frac{1}{2}}{\mapsto} n * \mathsf{S}(s,\pi \dot{-} \pi')) \vee (\neg b \implies s_\tau \overset{\frac{1}{2}}{\mapsto} x * \mathsf{S}(s,\pi' \dot{-} \pi)\}\end{array}} \quad [\textsc{CAtm}]$$

Figure 1: Thread-modular specifications of atomic operations

the states of the atomic integer with a fraction of the shared resource; and trans is a boolean predicate, encoding all the valid transitions that a particular instance of AtomicInteger can take.

Any thread calling a method of the AtomicInteger can acquire or release a fraction of the shared resource, depending on its role and the current state of the atomic integer. This is specified in AtomicIntegers contract. In order for a thread to be eligible to call a method, it has to possess a token indicating its role and the value of the atomic state upon its last visit. This is captured by an abstract predicate handle (line 7 of Listing 3). Essentially, this abstract predicate witnesses the role of the calling thread, its last seen value of AtomicInteger, and the fraction of the token.

The constructor of the AtomicInteger absorbs the resource associated with the initial value. Often, as seen e.g. in Semaphore, if the resource is obtained by a compareAndSet-based competition, then the synchroniser owns the resource at the beginning. But, if the threads start their life with some resources in their hands (such as the active threads in CountDownLatch), then the synchroniser does not own any resource in its initial step.

The get method exchanges the resources based on the view of the calling thread. In a competition-based synchronisation threads do not obtain any resource by calling the get method; they only update their knowledge about the current state. Apart from the thread's handle, any thread calling set(**int** v) has to provide (1) its permission to write the value v to the atomic integer, (2) the resources associated to its current view, and (3) the resources associated to the value v, *i.e.* the next state of the atomic integer. Upon return of the set method, the calling thread only obtains a handle updated with the thread's new view. Also the thread trying to atomically update the value of an atomic integer by calling compareAndSet(**int** x,**int** n) has to have the permission for the transition from x to n and the right handle to call this operation.

Following our formal specification (see Figure 1), our extension of the contract of AtomicInteger captures that the compareAndSet(**int** x, **int** v) method absorbs the *difference* between the resources that the synchroniser will hold in case of a successful update, *i.e.* the resources associated with n, and the resources that the synchroniser object currently holds, *i.e.* the resources associated with x. If the operation succeeds, the operation ensures the difference between the resources that the synchroniser

```
/*@given Set<role> rs;
given group (frac->group) inv;
given (role,int->frac) share;
given (role,int,int-> boolean) trans;  @*/
class AtomicInteger {
  private volatile int value;
/*@group handle(role r,int d,frac p);  @*/
/*@requires inv(share(S,v));   ensures (\forall* r in rs: handle(r,v,1));  @*/
  AtomicInteger(int v);

/*@given role r, int d, frac p;
requires handle(r,d,p) ** inv(share(r,d));
ensures handle(r,\result,p) ** inv(share(r,\result));  @*/
  public int get();

/*@given role r, int d, frac p;
requires handle(r,d,p) ** trans(r,d,v);
requires inv(share(S,v)) ** inv(share(r,d));
ensures handle(r,v,p);@*/
  public void set(int v);

/*@given role r, int m, frac p;
requires handle(r,x,p) ** trans(r,x,n)
requires inv(share(S,n)-share(S,x));
ensures  \result==> (handle(r,n,p) ** inv(share(S,x) - share(S,n));
ensures !\result==> (handle(r,x,p) ** inv(share(S,n) - share(S,x));  @*/
  boolean compareAndSet(int x, int n);
}
```

Listing 3: Contracts for AtomicInteger: Exclusive and Shared-reading

owned before the call, *i.e.* the resources associated on x, and the resources that holds after the successful update, *i.e.* the resources associated with n. If the operation fails, no resources are exchanged. Instead all resources specified in the pre-condition are returned. In the specification of AtomicInteger, the difference between the resources turns into the subtraction operation between two fraction types. The subtraction between two permissions is defined as zero if the result of the operation becomes negative. Besides, as explained above, inv(0) is equivalent to **true**. Therefore, as expressed in the contract of compareAndSet, the difference between the resources associated with the two states x and n determines if the calling thread releases or obtains fractions of the shared resource.

Essentially, our extension for the contract of AtomicInteger class is realised by: (1) defining the share function to map the fractions to the atomic state, and (2) updating the specification of compareAndSet with the cut-off subtraction between fractions. In the next section we demonstrate how one can use this specification of AtomicInteger to verify an implementation of a shared-reading synchroniser.

6 Verification

In this section we demonstrate how to verify the specification of a shared-reading synchroniser w.r.t. its implementation using an instance of AtomicInteger. For space reasons, we only explain the verifica-

```
/*@ given group (frac -> resource) rinv;  @*/
public class Semaphore{
 /*@ ghost final int num;     ghost Set<role> roles = {T};
 group initialized(int d,frac p) = sync.handle(T,d,p);
 resource held(int d,frac p) = initialized(d,p);
 group inv(frac p) = rinv(p);
 frac share(role r, int c){   return (r==S && c>=0 && c<num)?(c/num):0; }
 boolean trans(role r, int c, int n){
    return (r==T && c>0 && n==c-1) || (r==T && c<max && n==c+1); } @*/
 private AtomicInteger/*@ <roles,inv,share,trans> @*/ sync;

 /*@ requires rinv(1) ** n>0;    ensures initialized(n,1) ** num == n;   @*/
 Semaphore(int n){
 /*@ set num = n;  fold sync.inv(share(n));   @*/
   sync=new AtomicInteger/*@<roles,inv,share,trans>@*/(n);
 /*@ fold initialized(n,1); @*/
 }
}
```

Listing 4: Verification of Semaphore: constructor.

tion of Semaphore using the VERCORS tool set; the verification of CountDownLatch is very similar to Semaphore. Moreover, to show that our new specification still supports verification of exclusive access synchronisers, we have also verified an implementation of a SpinLock. All examples are available online [16, 22, 7]. The examples are automatically verified using the VERCORS tool set [27]. VERCORS is the tool that encodes our specified programs to intermediate languages like Viper [17] and Chalice [15] to be verified by permission-based SL back-ends like Silicon [17].

6.1 Semaphore: verification

Class Semaphore implements a synchroniser where a group of max permits threads simultaneously can have read access to a shared resource. The full code for this class is specified in Listing 4, Listing 5 and Listing 6.

The Semaphore class is parametrized with the resource invariant defined by its client program. The instantiated semaphore uses two predicates as tokens to detect if a thread holds a fraction of the shared resource: initialized and held.

An instance of a semaphore protects a shared resource with a specified maximum number of permits which is defined as a ghost variable within the class (line 3 of Listing 4). To instantiate an object of AtomicInteger, the Semaphore class has to define the required protocol. Resources are acquired using a compare-and-set based competition. All participating threads have identical roles in the specification. The shared resource to be protected by AtomicInteger is the same as the surrounding program passes on to the semaphore (Listing 4, line 6). The definition of the share function defines the fraction of the shared resource that must be held by AtomicInteger in each state (Listing 4, line 7). The definition given for the valid transitions expresses that in each update the difference between two states must be one unit (Listing 4, line 8).

```
/*@ given int d, frac p;
requires initialized(d, p) ** d<=num ** d>0;
ensures held(?w,p) ** rinv(1/num) ** w<num ** w>=0;@*/
  public void acquire(){
/*@ unfold initialized(d,p);   @*/
    boolean stop = false; int c = 0;
    while(!stop) { /*@ fold sync.inv(sync.share(T,d)); @*/
      c = sem.get();
      if( c > 0 ){    int nextc = c−1;
  /*@ fold sync.trans(T,c,nextc);
  fold sync.inv(sync.share(S,nextc)−sync.share(S,c)); @*/
        stop = sem.compareAndSet(c,nextc);
      }
    } /*@ fold held(nextc,p); @*/
  }
```

Listing 5: Verification of Semaphore::acquire().

6.1.1 Constructor

The client of the semaphore instantiates the object with a number of available units to acquire. Thus, it has to provide the resources associated with the initial value of the semaphore. After storing the maximum number of permits in the ghost field, the body of the constructor can feed the AtomicInteger class with the resources associated with its initial value (see line 14 of Listing 4). In return, the constructor of AtomicInteger returns its handle, which can be used to establish the postcondition of the constructor of the semaphore as defined in line 4. Finally, the semaphore ensures a full initialized token to the client program, which can be distributed in portions among the participating threads.

6.1.2 Methods

The annotated versions of the methods acquire() and release() are presented in Listing 5 and Listing 6, respectively. Having a fraction of the initialized token given by the client program, each thread is authorized to start its competition to acquire a permit of the shared resource protected by the semaphore. First, the acquiring thread has to read the current state of the atomic integer to see how many permits are still available. To achieve this, the body of the acquire has to unfold the provided initialized token to capture the required handle of the get method from AtomicInteger (line 5 of Listing 5). According to the provided protocol for the AtomicInteger, the thread does not have any resource associated with its view. Therefore, having the right handle suffices to read the current state of the sync object (see line 8 of Listing 5). To acquire a unit of the available permits the thread must decrement the current state by one. So it folds all the required abstract predicates as the specification of compareAndSet demands. Based on the given definition for the protocol, the acquiring thread does not need to provide any resources at this step. In case of a successful update, the compareAndSet(c,nextc) returns one unit of the shared resource, *i.e.* inv(1/num) (see 12 of Listing 5). The successful thread can then leave the body of acquire after folding the held predicate using the available handle from AtomicInteger. In the post-condition of the acquire method, ?w denotes the existence of a view for the calling thread after the call. Finally, if the thread fails to decrement the state, it continues reading the current state and trying to atomically decrement the state.

Releasing a fraction of the shared resource is symmetric to the acquire method. It should be easy to follow the reasoning steps presented in Listing 6. We only note here that the thread calling the release method provides the fraction of the shared resource it owns. Then, in an attempt to increment the current

```
/*@ given inr d,frac p;
requires held(d,p) ** rinv(1/num) ** d<num ** d>=0;
ensures initialized(?w,p) ** w<=num ** w>0 ;  @*/
  public void release(){
/*@ unfold held(d,p); unfold initialized(d,p); @*/
    boolean stop = false;
    while(!stop) {
      int c = sync.get();       int nextc = c+1;
/*@ fold sync.trans(T,c,nextc); @*/
/*@ fold sync.inv(sync.share(S,nextc)−sync.share(S,c)); @*/
      stop = sync.compareAndSet(c,nextc);
    } /*@ fold initialized(nextc,p); @*/
  }
```

Listing 6: Verification of Semaphore::release().

state of the atomic integer, if it succeeds it gives up the permit by folding the inv abstract predicate required by compareAndSet(c,nextc) at line 11 of Listing 6.

7 Conclusion and Related Work

Many different extensions of CSL are proposed in the literature. After RGSep [26] and Deny-Guarantee Reasoning [10], CAP [9] was introduced to reason about atomic operations. In CAP, resources are encoded together with the environment interference in an atomic rule to reason about synchronisation with finer granularity. The dream of having an universal logic for concurrent programs resulted in developing various extensions of CAP, namely HOCAP [24], iCAP [23] and, finally, Iris [14]. Iris is a PBSL based logic to reason about fine-grained concurrent data structures. It supports resource algebras, invariants and higher-order predicates. The user has to instantiate the logic with the elements of the target programming language. Currently, Iris-based verification is performed in Coq. Finally, Caper [8] is a verification tool where the core logic is based on CAP with additional features taken mainly from iCAP and Iris.

All the above mentioned works focus on the development of a generic, universal and powerful program logics. Instead, we treat reasoning about atomic operations at the specification level using an already existing logic, *i.e.* PBSL. We reuse an existing specification language (JML) to support more intuitive assertion language. This allows one to employ currently existing PBSL verifiers like Silicon [17] and Verifast [12]. We modified our approach in such a way that the new specification of AtomicInteger can be used to verify both exclusive and shared-reading synchronisers. This is done by defining a function that associates the atomic state to the fractions of the shared resources. The definitions of protocols and resource invariant are updated accordingly. Then, by proposing the cut-off subtraction operation in permissions we updated the specifications of atomic operations. We presented a set of mechanically verified examples to demonstrate how our new specifications can be used to verify implementations of synchronisers. In a separate work, we have presented how the specification of these synchronisation classes are used to verify a complete Java program [2].

8 Acknowledgments

The work presented in this paper is supported by ERC grant 258405 for the VerCors project.

References

[1] *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 2002.

[2] A. Amighi, S. Blom, S. Darabi, M. Huisman, W. Mostowski, and M. Zaharieva-Stojanovski. Verification of concurrent systems with vercors. In *Formal Methods for Executable Software Models - 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures*, pages 172–216, 2014.

[3] A. Amighi, S. Blom, and M. Huisman. Resource Protection Using Atomics - Patterns and Verification. In *APLAS*, pages 255–274, 2014.

[4] S. Blom and M. Huisman. The VerCors Tool for Verification of Concurrent Programs. In *FM*, pages 127–131, 2014.

[5] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In Palsberg and Abadi [19], pages 259–270.

[6] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis Symposium*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.

[7] Verified CountDownLatch: https://github.com/utwente-fmt/vercors/blob/master/examples/synchronizers/CountDownLatch.java.

[8] T. Dinsdale-Young, P. da Rocha Pinto, K. J. Andersen, and L. Birkedal. Caper - Automatic Verification for Fine-Grained Concurrency. In *ESOP*, pages 420–447, 2017.

[9] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In *ECOOP*, pages 504–528, 2010.

[10] M. Dodds, X. Feng, M. J. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *ESOP*, pages 363–377, 2009.

[11] A. Gotsman, J. Berdine, and B. Cook. Precision and the Conjunction Rule in Concurrent Separation Logic. *Electr. Notes Theor. Comput. Sci.*, 276:171–190, 2011.

[12] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NFM*, pages 41–55, 2011.

[13] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[14] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*, pages 637–650, 2015.

[15] K. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In *Lecture notes of FOSAD*, volume 5705 of *LNCS*. Springer, 2009.

[16] Verified Lock: https://github.com/utwente-fmt/vercors/blob/master/examples/synchronizers/ReentrantLock.java.

[17] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *VMCAI*, pages 41–62, 2016.

[18] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

[19] J. Palsberg and M. Abadi, editors. *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. ACM, 2005.

[20] M. Parkinson and G. Bierman. Separation logic, abstraction and inheritance. In *In POPL*, pages 75–86. ACM Press, 2008.

[21] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS* [1], pages 55–74.

[22] Verified Semaphore: https://github.com/utwente-fmt/vercors/blob/master/examples/synchronizers/Semaphore.java.

[23] K. Svendsen and L. Birkedal. Impredicative Concurrent Abstract Predicates. In *In ESOP*, pages 149–168, 2014.

[24] K. Svendsen, L. Birkedal, and M. J. Parkinson. Modular reasoning about separation of concurrent data structures. In *ESOP*, pages 169–188, 2013.

[25] V. Vafeiadis. Concurrent separation logic and operational semantics. *Electr. Notes Theor. Comput. Sci.*, 276:335–351, 2011.

[26] V. Vafeiadis and M. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR*, pages 256–271, 2007.

[27] Vercors tool set: http://ctit-vm3.ewi.utwente.nl/vercors-verifier/.