Technical Report

Resource Protection using Atomics:

Patterns and Verifications

Afshin Amighi Stefan Blom Marieke Huisman

University of Twente

{a.amighi,s.blom,m.huisman}@utwente.nl

Abstract

Modular reasoning about non-blocking concurrent data structures is crucial to establish the correctness of concurrent applications. To achieve this, specifications of the synchronization mechanisms used by these non-blocking concurrent classes to prevent concurrent access to shared data, are essential. This paper presents an approach to specifying such lock-free synchronization mechanisms in terms of the thread's role and permissions. The approach is formalized in a specification for the `AtomicInteger` class from the `java.util.concurrent` library, using abstract predicates and permission-based concurrent Separation Logic. The specification is set up to capture different synchronization patterns, both cooperative and competitive. We illustrate the use of the patterns in three case studies, where for each case study we verify that it indeed correctly synchronizes access to the protected data.

Categories and Subject Descriptors F.3.1 [*Theory of Computation*]: LOGICS AND MEANINGS OF PROGRAMS—Specifying and Verifying and Reasoning about Programs

General Terms Program Logic, Verification, Separation Logic, Non-blocking

Keywords concurrency, atomics, Java volatiles, separation logic, permissions, specification, verification

1. Introduction

To increase performance, modern processors employ multi-core architectures, so that extra computing power can be realized by parallel processing. To make optimal use of a multi-core architecture, the applications running on top of it should be *multi-threaded*. Typically, multi-threaded applications use synchronization to ensure there are no conflicting accesses to the same memory location, *i.e.*, to avoid data races. This is commonly done by using blocking synchronization techniques, such as monitors or mutex locks to protect shared resources. However, the use of blocking techniques can have a negative impact on performance, therefore non-blocking synchronization techniques are used more and more.

The main idea of non-blocking synchronization is to employ atomic operations – *atomic read*, *atomic write* and *atomic conditional update* (compare-and-set) – to encode a synchronization pattern, instead of using standard locking. Although the use of non-blocking algorithms can lead to a significant speed-up, they are also notoriously error-prone. Consequently, it is very important to provide the necessary (tool-supported) formal machinery to ensure their correctness.

Starting point and motivation for our work is the implementation of a lock-less hash table [10], especially designed for state space exploration in the multi-core model checker LTSmin [2]. Because of this application domain, its correct behaviour is essential. The hash table is used as a shared storage, containing all visited states. The Java implementation of this algorithm employs atomic classes from `java.util.concurrent.atomic` to implement a synchronization mechanism to protect the entries in the hash table.

Motivated by the hash table example, we investigate how correctness of multi-threaded programs can be established if classes from the `util.concurrent.atomic` library are used to protect access to shared resources. This paper addresses this question in a systematic way. We provide a classification of synchronization patterns using the `AtomicInteger` class. All patterns that can be used to ensure exclusive access are exemplified by means of a simple Java program. Moreover, we provide a specification of the `AtomicInteger` class that captures all these synchronization patterns.

The main idea behind our specification is that we consider the synchronizer (*i.e.* `AtomicInteger` in this

case) as a *permission manager*. Each thread that uses the synchronizer has a specific role, and depending on the role and the state of the thread, the permission manager grants and retains permissions to access the shared resource. This permission management protocol is declared abstractly as an argument of the synchronizer. The synchronizer's method contracts describe how permissions to access the protected resources are transferred from the synchronizer to the threads and back. When the specific synchronizer instance is created, concrete instantiations of the roles and protocols have to be provided, depending on the synchronization pattern that is actually needed. To describe the specifications and the predicates encoding the roles and the protocol, we use permission-based separation logic for Java [4, 8].

As mentioned above, we discuss three examples, each implementing one typical synchronization pattern. One of the examples (the `SingleCell` example) is a simplified version of the lock-less hash table [10], where instead of an array of elements, only a single element is stored in a shared location. For each synchronization pattern, we discuss how the role and protocol have to be defined. Moreover, for each example, we present a machine-checked correctness proof, showing that it indeed protects a shared resource, and avoids data races.

This paper is structured as follows: Section 2 provides the necessary background on the treatment of atomic variables according to the Java Memory Model, and on permission-based separation logic and our program specification language. Section 3 presents the different synchronization patterns and introduces three case studies using `AtomicInteger` as a synchronization primitive. Section 4 explains the specification of the `AtomicInteger` class. Section 5 discusses additional requirements that should be verified for the concrete instantiations of the roles and protocol predicates, to avoid duplication of permissions, and thus to ensure overall soundness of the approach. The correctness proofs of the examples are presented in Section 6. Finally, related work is described in Section 7, while Section 8 draws conclusions, and discusses future work.

2. Background

This section provides some necessary background information on the support provided by Java for the atomic treatment of variables, and on the specification language and verification technique that we use to formally specify and reason about program behaviour.

2.1 Volatiles and Atomics

To support thread-safe access to single variables, Java provides the `util.concurrent.atomic` package, as part of Java's general concurrency API. The `atomic` package contains a collection of classes that provide a wrapper for `volatile` variables with appropriate atomic operations for lookup, update, and conditional update. When a variable is declared volatile in Java, changes to the variable are immediately visible to other threads, *i.e.*, its value will never be cached thread-locally.

The behaviour of concurrent programs, including the use of volatile variables, is formally specified as the Java Memory Model (JMM) [12]. We briefly recapitulate some of its main characteristics that are necessary to understand the behaviour of volatile variables.

The JMM uses a *happens-before* relation to order memory events. If an event e_i happens-before e_j, then e_i is visible to (ordered before) e_j. Two accesses to the same variable are *conflicting* if at least one of them is a write access. If a given program contains two conflicting accesses that are not ordered by the happens-before relation, then the program contains a *data race*. A set of actions is *sequentially consistent* if: (1) there is a total order of the actions consistent with the program order; and (2) any read to a variable observes the last written value to the same variable. A program is considered *data race free* if in each sequentially consistent execution of the program, there is a happens-before relation between each pair of conflicting actions.

The JMM specifies that a write to a `volatile` field always is in the happens-before relation with every subsequent read of that field, therewith ensuring that conflicting accesses to `volatile` variables are never data races. Further, the JMM guarantees that the last written value of a `volatile` variable is always visible to all threads. This makes volatile variables suitable to implement synchronization mechanisms, where it is essential that threads have a consistent view on the synchronizer. In terms of caches, to ensure that a write to a `volatile` variable is visible to all the threads, after a write the local cache always is flushed to main memory. Similarly, to ensure that the latest value is read, the local cache is always invalidated before any `volatile` read.

The classes in Java's `atomic` package encapsulate the state of a `volatile` field, essentially providing three primitive operations: atomic read, atomic write and conditional atomic update (compare-and-set). This paper studies class `AtomicInteger`, which encapsulates a private volatile field of type integer. However, our techniques are also applicable to other atomic classes. `AtomicInteger` essentially provides the following methods: `get()`, returning the value that was last written to the volatile field; `set(int v)`, atomically assigning the value `v` to the volatile field; and `compareAndSet(int x,int n)`, atomically checking the current value and updating it to `n`, if it is equal to the expected value `x`, otherwise leaving the state unchanged, and then returning a boolean to indicate whether the update succeeded (the other methods in

the class are all variations of the methods described above).

2.2 Specification Language

To specify the behaviour of the `AtomicInteger` class and the example synchronization patterns, we use a variant of Separation Logic (SL). SL is an extension of Hoare Logic, originally developed to address the aliasing problem when reasoning about mutable data structure [19]. Key ingredients of the logic are the *points-to* predicate, denoted $l \mapsto v$, expressing that expresssion l points to a location on the heap, and this location contains the value v, and the *separating conjunction* operator, denoted $*$, expressing that two formulas are validated by disjoint parts of the heap. This allows implicit reasoning about two references not being aliased, *e.g.*, the assertion $l_1 \mapsto v_1 \ * \ l_2 \mapsto v_2$ states that l_1 and l_2 are two *disjoint* locations of the heap containing values v_1 and v_2 respectively. As we reason about Java programs, we use Parkinson's variant of SL for Java, where the expression pointing into the heap is a *field access of an object* [17].

The ability of SL to reason about the disjointness of the heap also makes it suitable to reason in a thread-modular way about multi-threaded programs, as shown by O'Hearn [13]. To make concurrent SL flexible enough to reason about multi-threaded Java programs, where simultaneous reads on the same location are allowed, it is combined with the theory of permissions [5] as *permission-based Separation Logic* [4]. Each points-to predicate is decorated with a fractional permission, *i.e.*, a value in the domain $(0, 1]$. At any point in time, each thread holds a set of permissions on locations. If a thread has a full permission for a certain location, *i.e.*, the value 1, then it has a *write* permission on this location. If a thread has a fractional permission, *i.e.*, a fraction less than 1, then it has a *read* permission on this location. The verification rules for field lookup and update require that a thread holds the appropriate permissions before performing the action. Soundness of the logic ensures that the total number of permissions on a location never exceeds 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. This in turn ensures that there are no data races in verified programs. Notice that for volatile variables, the JMM already ensures that there are no data races, so for volatiles, the distinction between read and write permissions is not necessary.

Permissions can be split and combined to change between read and write permissions. They can be transferred between threads upon thread *creation*, and also upon *joining* a terminated thread. Moreover, permissions can also be transferred by synchronization. For example, when data is protected by a lock, this is specified by associating the permission to access this data with the lock. This association is called the *resource invariant*. When a thread acquires the lock, it obtains these permissions; when it releases the lock, it has to give up these permissions [7]. This same approach is used in this paper for the synchronization mechanisms encoded using volatile variables.

Formally, the syntax of our expression language is defined as follows.

$$lop \in \{*, \&, |\} \qquad qt \in \{\exists, \forall\}$$
$$F \ ::= \ e \ \mid \ \texttt{PointsTo}(e.f, \pi, e) \ \mid \ F \ lop \ F \ \mid \ (qt \ T \ \alpha)(F) \ \mid \ \kappa$$

The assertion $\texttt{PointsTo}(e.f, \pi, v)$ corresponds to $e.f \overset{\pi}{\mapsto} v$ in traditional notation. It holds for a thread t if the expression $e.f$ points to a location on the heap that contains the value v and, in addition, the thread t has at least permission π to access this location. A formula $\phi_1 * \phi_2$ holds for a heap if the heap can be split into two *disjoint* heaps, and the first sub-heap satisfies ϕ_1, while the second sub-heap satisfies ϕ_2. Finally, assertions can also contain abstract predicates (κ) that encapsulate the state space [14]. Abstract predicate bodies are not visible outside their scope. In proof outlines, the abstract predicates should be explicitly opened when they are in scope, otherwise their body cannot be used.

In the specification below, we sometimes require the predicate to be a *group*. This means that the predicate can be split over permissions, see [7] for more details.

When the actual value stored is not important, we sometimes abbreviate the assertion `PointsTo` as `Perm`: $\texttt{Perm}(\texttt{x.f}, \pi) \overset{\text{def}}{=} \exists v.\texttt{PointsTo}(x.f, \pi, v)$. Notice that we have the following correspondence [15]: $\texttt{PointsTo}(x.f, \pi, v) \Leftrightarrow \texttt{Perm}(\texttt{x.f}, \pi) \ * \ \texttt{x.f} == v$, which we will use whenever appropriate.

Finally, when we are not interested in the actual fraction associated with a read permission (because it refers to immutable data), we use a special value ϵ. The *minimum required permission* to read `x.f` is defined as $\texttt{Perm}(\texttt{x.f}, \epsilon)$, satisfying the following axiom: $\texttt{Perm}(\texttt{x.f}, \epsilon) * \texttt{Perm}(\texttt{x.f}, \epsilon) = \texttt{Perm}(\texttt{x.f}, \epsilon)$

3. Synchronization via AtomicInteger

As mentioned above, because of the requirement that threads always see the most recent value of an atomic variable, this provides a flexible way to synchronize threads, which programmers can use instead of lock-based synchronization. In fact, lock-based and other synchronization mechanisms are typically implemented in terms of atomic variables. This section describes various patterns of state-based synchronization in terms of atomic integers.

In a shared memory concurrency setting, two kinds of thread interactions are distinguished: *cooperation* and

competition [18]. In a cooperative interaction, threads share a resource based on a predefined access protocol. In this interaction there is a *cooperative synchronizer* that coordinates the threads using the shared resource. In a competitive interaction, the resource is obtained by a thread that wins the competition. A *competitive synchronizer* runs the competition and provides the winner thread with the (temporary) access to the resource.

3.1 Synchronization Patterns

We first explain the different synchronization patterns that can be identified, using different combinations of primitive atomic operations from `AtomicInteger` (*i.e.*, `get`, `set` and `compareAndSet`)[1]:

1. `get` and `set` (GS): Atomic read and write can be used to implement a cooperative synchronizer. Every thread has a designated synchronizer state in which it obtains the resource, and all threads attempt to reach their designated state. When a thread writes the atomic integer, it implicitly signals who *should* own the resource next (cooperation). Based on the value written into the synchronizer, ownership of the resource is transferred to the appropriate thread waiting for that particular value. Producer-Consumer (also known as bounded buffer) is a well-known example that can be implemented this way.
2. `set` and `compareAndSet` (SC): Atomic write and conditional write can be used to implement a competitive synchronizer. Threads are competing to obtain the protected resource by calling `compareAndSet`. A thread that succeeds in changing the synchronizer state, obtains the resource. When it no longer needs the resource, it signals its availability by reverting the synchronizer state to the initial value (`set`). Threads failing to obtain the resource have to wait until the synchronizer state is reverted. Spin-lock implementations using `AtomicInteger` are a well-known example of this pattern.
3. `get` and `compareAndSet` (GC): Atomic read and conditional update can be used to implement a shared access synchronization mechanism. Typically, in such a case the value of the synchronizer state indicates the number of threads that are currently using the resource. Any thread trying to acquire or release the resource has to take a snapshot of the current synchronizer state using an atomic read operation, and then tries to signal the other threads by updating the synchronizer state using `compareAndSet`. `CountDownLatch` or `Semaphore` are well-known examples of this pattern. Besides, this pattern generally is used in lock-free data structures where `AtomicReference` coordinates the threads. As it cannot provide exclusive access[2], we do not discuss this pattern further.
4. `get`, `set` and `compareAndSet` (GSC): Atomic read, write and conditional update can be used to implement a competitive synchronizer. Threads compete with each other to obtain the resource by calling `compareAndSet`. A thread that succeeds in changing the state, obtains the resource. When it no longer needs the resource, it signals this by setting it to a new value. Threads that failed to obtain the resource have to wait until they see that the value of the synchronizer has changed, and then they can continue their job. The difference with the SC pattern above is that here the end of the critical section is signalled by setting the value to a new value, after which the synchronization behaviour might change (for example, only allowing read access to the protected resource). Below we will discuss an example where a shared resource is accessed using this pattern.

3.2 Synchronization Examples

Next, we present three examples where `AtomicInteger` is used to implement an exclusive access synchronization mechanism, illustrating patterns 1, 2 and 4. The pattern explained in case 3 is a common pattern for *shared access synchronization*. It is not in the context of our current work; however in future work we plan to generalise our approach to also reason about shared access synchronization.

ProducerConsumer (GS) To illustrate pattern 1, we discuss an implementation of a typical Single Producer/Single Consumer algorithm using atomic read and write operations to protect access to a single-state shared buffer. Lst. 1 shows two methods `produce` and `consume`, sharing a field `data`, that implement this algorithm. Typically, they will be executed as part of a surrounding loop.

The `AtomicInteger` denotes the state of the buffer, which is either full (`F`) or empty (`E`). Both the producer and the consumer wait until the buffer gets into the appropriate state. As soon as the state changes to the expected value, the waiting thread obtains the shared resource. When it is done, it changes the state, so that the other thread can access the resource.

SpinLock (SC) To illustrate the second pattern, Lst. 2 shows the implementation of a single-entrant spin-lock using `AtomicInteger`. The atomic integer value encapsulates the state of the lock: locked (`L`) or unlocked (`U`). A successful atomic update from `U` to `L` transfers the lock to the calling thread. Consequently, failing threads enter a busy-wait loop, until the lock is released again. To release the lock, the thread holding

[1] The same patterns exist for atomic variables of different types.

[2] Assuming that `compareAndSet` is not used instead of `set`.

```
public class ProducerConsumer{
  private final int E = 0, F=1;
  private AtomicInteger sync;
  private int data; // shared buffer
  ProducerConsumer(){ sync = new AtomicInteger(E); }

  void produce(){
        write(); // updates shared buffer
        sync.set(F);
        while(!sync.get() == E); }

  void consume(){
        while(sync.get()==E);
        read(); // reads new content
        sync.set(E); }
}
```

Lst. 1. ProducerConsumer

```
public class SpinLock{
  private final int U = 0, L=1;
  private AtomicInteger sync;
  SpinLock(){ sync = new AtomicInteger(U); }

  void lock(){ while(!sync.compareAndSet(U,L)); }
  void unlock(){ sync.set(U); }
}
```

Lst. 2. SpinLock

```
public class SingleCell{
  final private int E = 0, W=1, D=2;
  final private int PUT = 0, SEEN = 1, COLN = 2;
  private AtomicInteger sync;
  private int data;
  SingleCell(){ sync = new AtomicInteger(E); }

  int find_or_put(int v){
    if(sync.compareAndSet(E,W)){
      data = v;
      sync.set(D);
      return PUT;  }
    if(sync.get()!=E){
      while(sync.get()==W);
      if(sync.get() == D)
        if(data == v)  return SEEN;
        else  return COLN; }
  }
}
```

Lst. 3. SingleCell

the lock calls `set(U)`, which results in the next round of competition for the waiting threads. Lst. 2 shows the two main methods of the class: `lock`, to obtain the resource, and `unlock`, to release the resource.

SingleCell (GSC) To illustrate pattern 4, Lst. 3 shows the implementation of a `SingleCell` algorithm, which is a simplified version of the lock-less hash table [10]. It provides a single method that finds or puts a value in a shared storage. After the assignment, the stored value will be immutable. Initially, all threads are competing to assign their value. If a thread is successful in making the assignment, it will report its success (return value `PUT`). All other threads have to wait until the value is assigned, and then they compare with the stored value. If the value in the cell is equal to the value the thread holds, it will return the value `SEEN`, otherwise it will signal that the cell contains a different value (return value `COLN`, for collision).

The atomic integer can contain 3 different values, denoting the different stages in the algorithm. Initially, the storage is empty (`E`). If a thread succeeds in obtaining writing access to the resource, the states becomes `W`. After the assignment, this thread changes the state to `D` to indicate it is done writing the value in the cell. Since the assigned value is intended to be immutable, the state cannot revert back to a previous value. Thus, any thread that reads the `W` state has to wait until it visits `D`, and then it can access the shared resource.

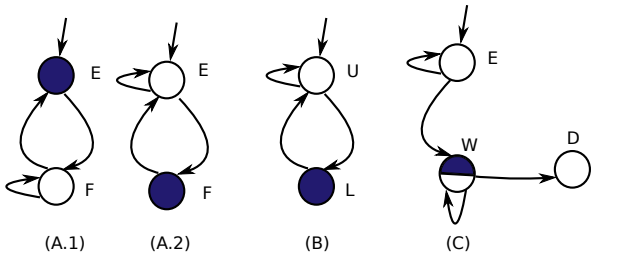

Figure 1. State machines of (A) `ProducerConsumer`, (B) `SpinLock` and (C) `SingleCell`

State Machines Finally, Figure 1 presents state machines that correspond to the examples above. In Figure 1(A) the state changes for the `ProducerConsumer` algorithm is shown, where A.1 shows the producer and A.2 shows the consumer; `SpinLock` is shown in Figure 1(B); and Figure 1(C) shows the state machine for `SingleCell` storage. Note that a coloured state indicates that a thread has *exclusive access*. The half-coloured state in (C) corresponds to the state where different threads have a different *interpretation* of the state: the thread that succeeded to update the state into `W` has exclusive access to the resource, while the other threads have no access to the resource.

4. Specification of AtomicInteger

Next, we specify the behaviour of the `AtomicInteger` class as an exclusive-access synchronization primitive. This specification describes how the permissions to access the shared resource are distributed among the different threads. Essentially, this is described as method contracts of the methods `get`, `set` and `compareAndSet`, where the preconditions specify the permissions that the invoking thread has to hand over to the synchronizer, while the postconditions specify the permissions that the synchronizer passes on to the invoking thread. We should stress here that the methods are considered as primitive operations, and the specification is not concerned with the correctness of their implementation.

4.1 Predicates and Parameters

To make a single specification of AtomicInteger that can capture synchronization patterns 1, 2 and 4 (*i.e.*, all exclusive access patterns) described above, the specification is parametrized by:

- a set of roles, *i.e.*, abstract description of the intended behaviour of the threads participating in the synchronization;
- a resource invariant, specifying the resources that are protected by the synchronizer;
- an abstract predicate trans, encoding the permitted transitions of state machine of the particular synchronization mechanism (cf. Figure 1); and
- a function share, specifying the fraction of the resource invariant that is transferred between the synchronizer and the invoking thread in this step.

In addition, AtomicInteger declares a predicate handle that is passed around as a token to threads that participate in the synchronization, which they use to prove that they are allowed to invoke an action.

Before discussing the full specification proposed for AtomicInteger, we first describe these ingredients in more detail.

Roles In a parallel execution, different threads play different roles. Essentially, a role is an abstraction that describes which operations and privileges are meaningful for a thread. Consider for example the two threads participating in the ProducerConsumer example: the producer stores elements in the buffer when it is empty, and the consumer reads elements from the buffer when it is full. This is an example of a cooperative synchronization mechanism, where all participating threads have different roles. The SpinLock example and the SingleCell example above are competitive synchronization mechanisms. In such a schema, all threads have the same role, *i.e.*, they try to obtain the resource whenever it is available. In fact the synchronizer as a globally known role coordinates the threads. Therefore we define a special role for the synchronizer, declared as a publicly visible constant in class AtomicInteger, to hold the resource when it runs the competition. The synchronizer passes the resource to the winner thread, and if the winning thread releases the resource, it gives it back to the synchronizer.

In our case studies, we denote the roles of the producer and the consumer with P and C, respectively. In the competitive examples, where all the threads have the same role, T abstracts the threads, and S is defined as a globally known role for the synchronizer.

Resource Invariant The shared resources that are protected by the synchronization mechanism are described by the *resource invariant*. This is similar to specifications of locking mechanisms [7, 13], where each monitor or lock is associated with a resource invariant.

However, because of pattern 4, where eventually all threads gain read access to the shared resource, in our specification, the resource invariant cannot be an arbitrary predicate, but it has to be a group, *i.e.*, it should be splittable over resources.

In the SpinLock example, the class will be parametrized with the resource invariant. The methods lock() and unlock() exchange this resource invariant between the client thread and the underlying synchronizer. In ProducerConsumer and SingleCell the field data is modelling the shared resource, and we will have the following resource invariant for these two examples:

```
group inv<frac p> = Perm(data,p);
```

Allowed Transitions The preconditions of the state changing methods in AtomicInteger, *i.e.*, set and compareAndSet operations, require that the state change of the synchronizer is allowed according to the state machine of Figure 1. This state machine is encoded in the trans predicate. Such an encoding will ensure for example that in SingleCell, no thread is allowed to change the state from W to E.

The trans predicate expects as arguments the role of the invoking thread, the current and the intended update state of the synchronizer.

In ProducerConsumer, the predicate is defined as:

```
pred trans<role r,int c,int n>=
  (r == P && c == E && n == F) ||
  (r == C && c == F && n == E);
```

In SpinLock the predicate is defined as:

```
pred trans<role r,int c,int n>=
  (c == U && n == L) || (c == L && n == U);
```

Finally, SingleCell has the following definition of trans:

```
pred trans<role r,int c,int n>=
  (c == E && n == W) || (c == W && n == D);
```

Permission Sharing The function share is used to define the fraction of the resource invariant that is transferred between the synchronizer and the invoking thread. In the specification, it is always used as an argument to the resource invariant predicate. In a precondition, it specifies the fraction of the resource invariant that has to be given up by the invoking thread, and that is passed back *into the synchronizer*. In a postcondition, it specifies the fraction of the resource invariant that is passed from the synchronizer *to the invoking thread*. Notice that since threads can only exchange permissions with the synchronizer, this ensures that any permission transfer passes *through* the synchronizer.

The share function is parametrized by a role, and the value of the atomic integer[3]. When share is used in the precondition, it gets as argument the value of the AtomicInteger that was last seen by the invoking thread, which might not necessarily be its actual value. This can be understood as follows: the share on the resource invariant that the invoking thread holds, depends on the value it last saw, and this has to be transferred from the invoking thread into the synchronizer. When share is used in the postcondition, it gets as argument the new value of the atomic integer. The new value thus decides the share of the resource invariant the thread obtains when the method is finished. However, there is one exception to this: when a conditional update fails, nothing is learned about the new value of the atomic integer, and instead the thread simply gets the resources related to the last known value back.

In a competition, the exclusive access is always won by successful compareAndSet operation. This means, the winner successfully visited the *expected* value. The special globally known role defined for the synchronizer, *i.e.*, S, holds the resource for the expected value. Therefore, in addition to the share specified by the standard role of the thread, the postcondition of a successful compareAndSet operation also returns a share of the resource invariant related to the special S role. This share is related to the expected value that was passed to the compareAndSet operation, and not the new value. This can be understood as follows: if the compareAndSet operation is successful, the expected value is the same as the value that was held in the atomic integer, and thus it should transfer the resources associated to this value that were stored inside the synchronizer. Moreover, the operations that can (potentially) change the state of the atomic integer, *i.e.*, the constructor, set and compareAndSet all require the invoking thread to return the share associated to S into the synchronizer.

As share might return 0 to indicate *no permissions* have to be transferred, we formally include 0 in the fractional permission interval, and we interpret inv<0> as the empty resource, *i.e.*, $\texttt{inv<0>} \stackrel{\text{def}}{=} \texttt{true}$.

The trans predicate and share function together describe the complete synchronization protocol. We provide the definition of the share function on our case studies.

ProducerConsumer (GS) For this example, the function share is defined as follows:

$$\texttt{share(P,E)} = 1 \quad \texttt{share(C,F)} = 1$$

Implicitly, in all other cases, the function returns 0. This specifies that the producer (with role P) obtains (via the postcondition of the get operation) full access to the shared buffer when it sees that the buffer is empty, and when it sets it to full, (via the precondition of the set operation) it gives up access. If later, it tries to obtain access again, by calling the get method, the precondition requires share(P,F) of the resource invariant, *i.e.*, no permissions, because F is the last value seen by the producer. Whenever get returns F, no permissions are obtained; only when the get operation returns E, the producer obtains full access again. For the consumer, a similar explanation applies.

SpinLock (SC): This is an example of a competitive synchronization mechanism, where the special S role is relevant. We define the function share as follows:

$$\texttt{share(S, U)} = 1$$

All other cases implicitly are 0, including all cases for the non-S role. This means that if a thread succeeds in the compareAndSet operation, changing the set from U to L, this thread obtains the full resource invariant (via the postcondition of a successful compareAndSet operation). Moreover, when the thread sets the state back to U, the precondition of the set operation ensures that the invoking thread has to give up access to the resource invariant, and hand it back into the synchronizer.

Notice that an alternative implementation, where an unnecessary compareAndSet operation would be used instead of a set operation to release the lock, would also correctly transfer permissions (because of compareAndSet's precondition).

SingleCell (GSC): This is again an example of a competitive synchronization mechanism. However, in this example, threads can also obtain a fraction of the resource invariant by using their normal role. The function share is defined as follows:

$$\texttt{share(S, E)} = 1 \quad \texttt{share(S, D)} = \epsilon \quad \texttt{share(T, D)} = \epsilon$$

All other cases implicitly are 0. This can be understood as follows. If the cell is still empty (state E), and a thread wins the competition for exclusive access, it obtains full access to the shared resource. When it is done writing the cell, and sets the state to D, it will hand the minimum fractional share on the resource invariant back to the synchronizer. Thus, implicitly, the thread will also keep a fractional share and thus keep read access to the cell. Once the state is set to D, any non-winning thread (*i.e.*, with role T) that sees this state change will obtain the minimum fractional share on the resource invariant (by means of the postcondition of get), and thus obtain read access on the cell.

[3] To completely follow this explanation, the reader might want to have a sneak preview of the specification of AtomicInteger in Lst. 5 on Page 9.

Notice that this transfer of permission relies on the fact that the state cannot change back from W to E, or from D to W or E. If this would be possible, more advanced specifications are necessary, allowing temporary shared access. As mentioned above, it is future work to generalize our specifications in this respect.

Handle To show that a thread is allowed to invoke an operation from AtomicInteger, the precondition demands the invoking thread to prove that it is allowed to call this operation given the role and the last seen state of the synchronizer. For this purpose, the AtomicInteger specification defines a special token, called handle, which can be used to prove that a thread has the right to invoke an action. The postconditions ensure that appropriate new handles for new actions are handed out to the invoking thread.

The handle is parametrized by the role of the calling thread and the last observed state of AtomicInteger. Any instance of a synchronization mechanism is associated with a particular set of threads. Therefore any thread (1) without a handle (*i.e.*, outside of the coordinated threads), (2) with an incorrect role, or (3) with a visited value that is outside of the synchronizer's reachable states, will therefore not be able to interfere with the threads that participate in this synchronization.

Notice that the use of handles is necessary to avoid that new resources can be created out of the blue. In particular, it avoids that get operations can be called multiple times to obtain multiple resources. Consider for example the incorrect variant of the producer from the ProducerConsumer example in Lst. 4. If we do not use handles in our specification, this method can be verified. The leakProducer() can start assuming its last observed state is F. Then, if the get operation returns E, it obtains the full resource invariant. Since share(P,F) is 0, we can freely add inv<share(P,F)> to the current state knowledge. But this means that get can be invoked again, and as a result, the thread obtains its second full share of the resource invariant, and thus it actually has obtained two write permissions on the buffer.

The use of the handle predicate avoids this problem. After a get operation, the thread obtains a handle capturing its current knowledge about the state of the synchronizer. When it invokes the get method, it has to provide this handle, and in addition also the resource invariant associated to this value.

Handles are specified as groups, *i.e.*, a predicate that can be split over permissions. This allows a thread to pass a fraction of its handle to a newly created thread. At the initialization of the AtomicInteger, the constructor issues a full handle for all roles that are passed to the synchronizer. These full handles are all given back to the thread that created the AtomicInteger. These full handles may then be split and passed on to any other thread participating in the synchronization.

```
public class ProducerConsumer{
  private int data;
  /*@ ... *@/ // invariants, predicates and protocols
  //@ group inv<frac p> = Perm(data,p);
  // an incorrect producer leaks the resource
  void leakProducer(){
    {true}  : close inv<share(P,F)>
    {inv<share(P,F)>}
   if(sync.get()==E)
    {inv<share(P,E)>*true}  :  close inv<share(P,F)>
    {inv<share(P,E)>*inv<share(P,F)>}
    if(sync.get()==E)
      {inv<share(P,E)>*inv<share(P,E)>}  :  open all
      {Perm(data,1)*Perm(data,1)}
  }
}
```

Lst. 4. Contracts without handles: An incorrect producer leaks the resource

Moreover, the fraction carried inside the handle correlates with the fraction that the share, based on the thread's role, assigns. Again we can imagine an incorrect code that splits the handle. Then using each portion calls the get twice and as a result obtains the full share twice. If a thread splits its handle, its share also must be split. To avoid duplication of permissions, therefore the share of the resource invariant that is returned is multiplied by the permission carried in the handle. In fact, handles are used as an evidence of the calling thread's *correct* knowledge. Thus, in the presence of handles the thread has no way to "fake" its knowledge.

4.2 Specification

Finally, we are ready to discuss the full specification of class AtomicInteger as a synchronizer for exclusive access. Listing 5 shows the complete specification. The parameters and predicates have been discussed above, here we only discuss the method specifications.

Constructor The constructor requires the share of the resource invariant associated to the special S role for the initial value of the atomic integer. These are the resources that are initially stored inside the synchronizer, and that can be won by the winning thread in a competitive synchronization mechanism. Notice that in a cooperative synchronization mechanism, the resources initially are supposed to be with one of the threads, and the synchronizer is only used to pass the resources on to the next thread.

The postcondition of the constructor provides handles for all roles (except the S role) that are involved in the synchronization, which can be split and passed to all threads that want to access the shared resource.

Method Get The precondition of the get method requires the handle corresponding to the last value (o) the invoking thread has seen (handle<r,o,p>), and it requires the thread to give up its corresponding share

of the resource invariant (inv<share(r,o).p>). It returns a handle for the result of the get operation (handle<r,\result,p>), together with the appropriate share of the resource invariant, *i.e.*, share(r,\result) multiplied by the fraction p in the handle that was provided.

Method Set The precondition of the set method requires the handle corresponding to the last value the invoking thread has seen, and it requires the thread to give up its corresponding share of the resource invariant. Moreover, it requires that this thread is allowed to change the state to the new value v. Last, it requires that the share related to the special S role of the new value is passed into the synchronizer.

The postcondition ensures that a handle for the newly written value v is returned, together with the appropriate share of the resource invariant.

Method CompareAndSet The precondition of the compareAndSet method requires the handle corresponding to the last value the invoking thread has seen, and it requires the thread to give up its corresponding share of the resource invariant. Moreover, it requires that this thread is allowed to change the state from the expected value x to the new value n. Last, it requires that the share related to the special S role of the potential new value is passed into the synchronizer.

The postcondition ensures that if the conditional update is successful, the thread obtains a handle for the newly written value n, together with the appropriate share of the resource invariant. Moreover, any share of the resource invariant associated with the special S role for the expected value are returned to the caller. If the conditional update was not successful, the thread obtains back all permissions it was forced to give up by the precondition, including its original handle.

5. Sound Permission Distribution

As explained, AtomicInteger uses function share to manage the permissions on the shared resource. This function defines which fraction of the permissions on the shared resource are held by the participating threads. For soundness, the definition for share is vital, as it should not allow the synchronizer to invent permissions. An incorrect definition may result in a state in which the total sum of the permissions held by the threads for a resource exceeds the full permission. Therefore, we need to define an extra check that ensures that the definition of share cannot create new permissions.

Let us first look at this problem intuitively. In the ProducerConsumer example, when the AtomicInteger is in state E we have two cases:

1. The producer is writing to the resource and the consumer is waiting. Here, both the producer and the consumer agree on the correct state. So we can check that sum of share(P,E), share(C,E) and share(s,E) does not exceed 1.
2. The producer has changed the state into F but the consumer, in its waiting loop, has not yet observed the new state, *i.e.*, it believes the state to be E. So: share(P,F) + share(C,E) + share(S,F) ≤ 1

```
class AtomicInteger
  /*@<  roles: set of role;
         inv: frac -> group;
         share: role, int -> frac;
         trans: role, int, int -> pred; >@*/ {

  private volatile int value;
  //@ group handle<role r,int o,frac p>;

  /*@
  requires inv<share(S,v)>;
  ensures \forall* r in roles: handle<r,v,1>; @*/
  AtomicInteger(int v);

  /*@
  requires handle<r,o,p>*inv<share(r,o)·p>;
  ensures handle<r,\result,p>*inv<share(r,\result)·p>; @*/
  public int get();

  /*@
  requires handle<r,o,p>*inv<share(r,o)>;
  requires trans<r,o,v>*inv<share(S,v)>;
  ensures handle<r,v,p>*inv<share(r,v)·p>; @*/
  public void set(int v);

  /*@
  requires handle<r,o,p>*inv<share(r,o)>;
  requires trans<r,x,n>*inv<share(S,n)>;
  ensures  \result==>
     (handle<r,n,p>*inv<share(r,n)·p>*inv<share(S,x)>);
  ensures !\result==>
     (handle<r,o,p>*inv<share(r,o)>*inv<share(S,n)>); @*/
  boolean compareAndSet(int x, int n);
}
```

Lst. 5. Contracts for AtomicInteger

In both cases the sum of the share of the permissions assigned to the producer, the consumer and the synchronizer is less than 1. Similar reasoning can be used to show that this property also holds when both threads see that the buffer is full. Thus, in ProducerConsumer in all the *reachable states*, the sum of the shares held by the threads does not exceed 1.

In general, the share function has to satisfy the following property: *in any snapshot of the execution, the sum of the fractions assigned to all the threads and the synchronizer must not exceed 1.* To show that this proof obligation is respected, we use the definitions of share and trans. From this, we first extract for each role the *maximal state machine*, which shows *all possible transitions* that a role can take, considering the allowed predicate. We then construct the product of the maximal state machines, and use this to reason about the sum of the shares for each *feasible snapshot*. Before illustrating this approach for our case studies, we introduce some notation. Nodes are labelled as r_v^π to denote that a thread with role r assumes value v as its last ob-

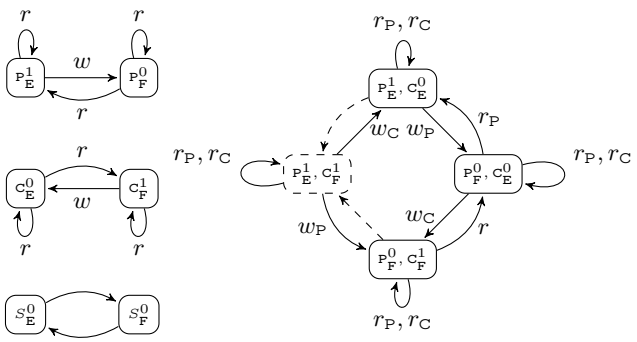

Figure 2. Possible states for `ProducerConsumer`

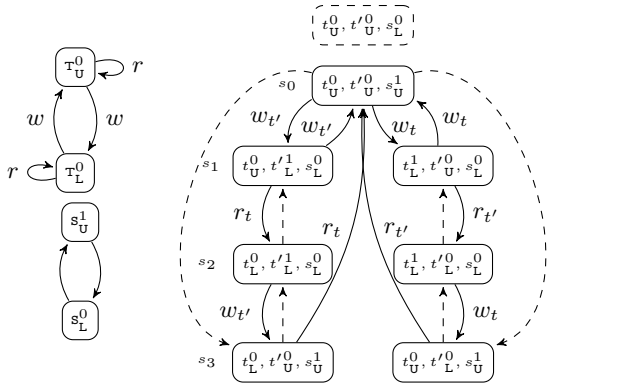

Figure 3. Possible states for `SpinLock`

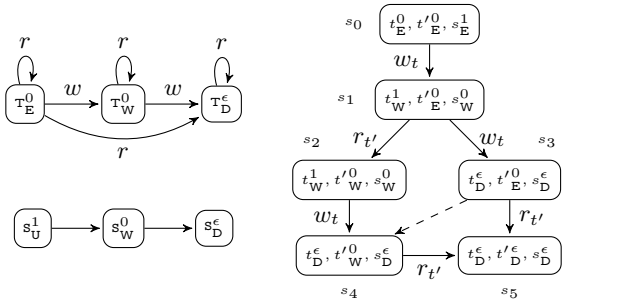

Figure 4. Possible states for `SingleCell`

served state; and, based on function `share`, holds fraction π. Transitions are labelled w, denoting a *write* to the synchronizer, and r, indicating a *read* of state of the synchronizer. The transitions of the special role `S` (*i.e.*, the synchronizer) are not labelled, since these transitions are due to received method calls. Finally, dashed transitions and nodes indicate impossible actions and states, respectively.

ProducerConsumer (GS) In this case each role is associated with a unique thread. Figure 2 shows the maximal model for each thread and the product of these models. Based on the `trans` predicate, the producer is the only role that can update the state from `E` to `F`. If it does so, its last observed state must change to `F`. Therefore, in the product, the state $(p^1_{\mathtt{E}}, c^1_{\mathtt{F}})$ is not reachable from $(p^1_{\mathtt{E}}, c^0_{\mathtt{E}})$. Similarly, it is not reachable from $(p^0_{\mathtt{F}}, c^1_{\mathtt{F}})$ because the definition of `trans` does not allow the producer to modify the state from `F` to `E`.

In the *feasible product of maximal state machines*, we then simply check for each state that the sum of the shares does not exceed 1, which holds in our example.

SpinLock (SC) As explained above, this is a competitive synchronization, where all threads have the same role. The `share` function specifies that initially the synchronizer holds the full share, and any thread that successfully calls `compareAndSet` obtains this full share.

In this case we construct the maximal feasible model for *two arbitrary threads*. Figure 3 depicts the maximal models for role `T`, the synchronizer with special role `S` and the product of the models for two arbitrary threads t, t' and an instance of the atomic integer s. To keep the picture simple we omitted self-loops.

First, at least one of the threads must agree on the state with the synchronizer. If the synchronizer does not agree with any of the threads about the current state, like $(t^0_{\mathtt{U}}, t'^0_{\mathtt{U}}, s^0_{\mathtt{L}})$, we can exclude this from the model. Secondly, any write modifying the initial state, transfers the full invariant to the updating thread. As a consequence, based on the contract of `set`, the other thread cannot have permission to update the state. In Figure 3, if thread t' succeeds to update the state to `L`, *i.e.*, $s_0 \xrightarrow{w_{t'}} s_1$, then it is the only one that has the permission to do the subsequent update to `U`. However, the transition $s_1 \xrightarrow{r_t} s_2$ is a valid transition because t can update its view on the current state by the `get` operation. The transition $s_2 \to s_1$ is invalid because if the atomic integer is not changed there is no way t can change its view. The transition $s_2 \xrightarrow{w_{t'}} s_3$ shows that t' uses its privilege to update the synchronizer while t still has not updated its view. Then t can observe the updated value in a transition from s_3 to s_0. Finally, the transition $s_0 \to s_3$ is invalid because t cannot change its view without any update to the synchronizer. The other transitions in the picture are just the symmetric variant of what we explained.

In the constructed feasible maximal model, we can check that in all the reachable states, the sum of the shares assigned to the threads and the synchronizer equals 1. This proves that with our definition of `share`, the atomic integer *neither invents nor looses* resources.

SingleCell (GSC) To prove soundness of the `share` function in `SingleCell` we show the reachable states for two arbitrary threads and an instance of the synchronizer in Figure 4. For clarity, the self-loops and unreachable states are not shown. Moreover, we just show the subset of state where thread t is the winning thread.

As can be seen, restricting w.r.t the one way protocol of the atomic integer prunes many traces that are not feasible in the execution. Further, the protocol allows t to update the state (s_0) from E to D, and to obtain the resource (s_1). As long as t holds the resource, t' can only update its view by reading the state: $s_1 \xrightarrow{r_{t'}} s_2$, $s_3 \xrightarrow{r_{t'}} s_5$ and $s_4 \xrightarrow{r_{t'}} s_5$. The reading in $s_3 \rightarrow s_4$ is not valid since t' is observing an incorrect state by this read operation. All other transitions labelled with w_t, interpreted as the write action taken by t, are valid transitions based on the contracts and definitions provided for the protocols. Finally, from the feasible maximal model, it is easy to see that the sum of shares never exceeds the full share.

6. Verification

This section discusses how the contract for the class AtomicInteger is used to prove correctness of our case studies, *i.e.*, to prove that they correctly synchronize access to the shared resource. For each case study we show the annotated program, with the complete proof outline. When the instance of the atomic integer class is created, it receives concrete instantiations of the abstract predicates and functions as described in Section 4.1. In addition, there is a global constraint stating that no thread can play more than one role. Notice that we assume that every thread has a fixed role. Extending our approach for dynamic roles is future work.

All annotated programs are verified with our VerCors tool set. This tool set is currently being developed to reason about multithreaded Java programs annotated with permission-based Separation Logic. The tool leverages existing verification solutions to multithreaded Java programs, by encoding verification problems into the Chalice language [11]. The Chalice verifier is then used to prove the translated program correct w.r.t. its specification. The examples as shown here are verified automatically, after providing a few additional proof hints in terms of intermediate state annotations that we left out here for clarity of presentation.

6.1 ProducerConsumer

Lst. 6 provides the specification and implementation of class ProducerConsumer, with appropriate proof outlines. For convenience, we repeat the specification arguments for the creation of the AtomicInteger synchronizer.

```
group inv<frac p> = Perm(data,p);
pred trans<role r,int c,int n>=
      (r == P && c == E && n == F) ||
      (r == C && c == F && n == E);
frac  share(role r,int s) {
      if(r == P && s == E) return 1;
      if(r == C && s == F) return 1;
      return 0;}
```

We do not discuss in detail how the shared buffer is actually accessed, we simply assume we have methods write and read that both require and ensure Perm(data,1).

We first discuss verification of the constructor. The constructor implicitly starts with the actual creation of the object. After this creation, the thread that executes the constructor has full access to all fields of the object, thus in particular, it has a write permission to data. To match the precondition of the AtomicInteger constructor to create the synchronizer object, we need to provide a predicate inv<share(S,E)> (where E is the initial value of the synchronizer). From the definition of share we can see that this corresponds to inv<0>, *i.e.*, the empty resource. Thus, we can close this predicate immediately. Framing tells us that after the call to the AtomicInteger constructor, Perm(data, 1) still holds. Moreover, from the postcondition of the AtomicInteger constructor, we obtain handles on E, both for the producer, and the consumer. We can close the inv predicate, and conclude that the postcondition holds.

The precondition of produce requires that this method is called by a thread that holds access to the shared buffer. Therefore, the call to method write is allowed, and the thread keeps its permission to access data. Logical reasoning allows us to conclude that trans(P,E,F) holds. Moreover, since share(P,E) = 1, we close Perm(data,1) into the resource invariant. Finally, since share(S,F) equals 0, we can also derive inv<share(S,F)>. This annotation matches the precondition of the call sync.set(F). From the postcondition, we can conclude inv<share(P,F)>. Next, to verify the while loop, we provide the loop invariant in line 24. To show how the loop is verified, probing the synchronizer we explicitly store the result of the get in a local variable v initialized with F. It is easy to see that this loop invariant can be established from. To see that the loop invariant is preserved, we consider the specification of get. In each loop iteration before the call to the get, v is F, so the loop invariant establishes the precondition of get. After the call, the handle and share is updated with the result of the invocation, which is stored in v. Finally, it should be noted that the loop only terminates when get returns E, and from the loop invariant we can conclude that in that case inv<share(P,E)> holds, which together with the modified handle implies the postcondition of produce.

Verification of the method consume is similar, except that it assumes that the method is invoked without holding any permissions.

6.2 SpinLock

To verify the SpinLock example, given the resource invariant associated with the lock, the instantiation of the atomic integer class parameters is defined as:

```
public class ProducerConsumer{
  ... // fields, invariants, predicates and protocols
  //@ ensures inv<1>∗handle<P,E,1>∗handle<C,E,1>;
  ProducerConsumer(){
    {Perm(data,1)*true} : close inv<share(S,E)>
    {Perm(data,1)*inv<share(S,E)>}
    sync = new AtomicInteger(E);
    {Perm(data,1)*handle<P,E,1>*handle<C,E,1>}
    : close inv<share(P,E)>
    {inv<1>*handle<P,E,1>*handle<C,E,1>}
  }
  //@ requires handle<P,E,1>∗inv<1>;
  //@ ensures handle<P,E,1>∗inv<1>;
  void produce(){
    {handle<P,E,1>*inv<share(P,E)>} : open inv<share(P,E)>
    {handle<P,E,1>*Perm(data,1)}
    write(); // updates shared buffer
    {handle<P,E,1>*Perm(data,1)}
    : close inv<share(P,E)>, trans<P,E,F>, inv<share(S,F)>
{handle<P,E,1>*inv<share(P,E)>*trans<P,E,F>*inv<share(S,F)>}

    sync.set(F);
    {handle<P,F,1>*inv<share(P,F)>}
    int v = F;
    //@ loop_invariant: handle<P,v,1>∗inv<share(P,v)>;
    while(v!=E){
      {handle<P,F,1>*inv<share(P,F)>}
      v=sync.get();
      {handle<P,v,1>*inv<share(P,v)>}
    }
    {handle<P,E,1>*inv<share(P,E)>}
    {handle<P,E,1>*inv<1>}
  }
  //@ requires true;
  //@ ensures true;
  void consume(){...}
}
```

Lst. 6. Annotated ProducerConsumer

```
public class SpinLock/∗@< resinv: frac −> group; >@∗/{
  ... // fields, invariants, predicates and protocols
  //@ requires resinv<1>;
  //@ ensures handle<T,U,1>;
  SpinLock(){
    {inv<1>}
    {inv<share(S,U)>}
    sync = new AtomicInteger(U);
    {handle<T,U,1>}
  }
  //@ requires handle<T,U,f>;
  //@ ensures handle<T,L,f>∗resinv<1>;
  void lock(){
    boolean stop = false;
    while(!stop){
{handle<T,L,f>*trans<T,U,L>*inv<share(T,U)>*inv<share(S,L)>}

      stop = sync.compareAndSet(U,L);
{!stop ==> handle<T,U,f>*inv<share(T,U)>*inv<share(S,L)>}

{stop ==> handle<T,L,f>*inv<share(T,L)·f>*inv<share(S,U)>}

    };
{handle<T,L,f>*inv<share(T,L)>*inv<share(S,U)>}
{handle<T,L,f>*inv<1>}
  }
  //@ requires handle<T,L,f>∗resinv<1>;
  //@ ensures handle<T,U,f>;
  void unlock(){
{handle<T,L,f>*inv<1>}
{handle<T,L,f>*trans<T,L,U>*inv<share(T,L)>*inv<share(S,U)>}

    sync.set(U);
{handle<T,U,f>*inv<share(T,U)>}
{handle<T,U,f>}
  }
}
```

Lst. 7. Annotated SpinLock

```
group inv = resinv;
pred trans<role r,int c,int n> =
     (c==U && n==L) || (c==L && n==U);
frac share(role r,int s){
          return (r == S && s == U) ? 1 : 0;  };
```

Lst. 7 shows the proof outline for this implementation. The SpinLock will protect its given resource invariant using the underlying AtomicInteger. Notice how the specification of the AtomicInteger ensures the standard lock specification: the contract of lock() states that a calling thread will receive full ownership upon successfully locking, while the contract of unlock() specifies that upon releasing the lock, ownership should be given up completely.

Compared to the original SpinLock example in Lst. 2, there is a minor change in the implementation of the lock method: we use a local variable stop to store the result of the compareAndSet action. This does not change the behaviour, but allows us to show more explicitly how the method is verified. Besides, the proof hints and loop invariants are omitted.

The client of the SpinLock feeds the constructor with the resource invariant. Holding resinv<1>, SpinLock establishes inv<share(S,U)> and feeds resinv<1> to the synchronizer to initialize the competition. In return, a handle for role T and state U with a full permission is returned. The main thread can split this handle and passes the fractional handles to the participating threads, so they can call the lock method.

The proof outline in lock shows how the SpinLock transfers full resource invariant to the successful thread. The thread holding the lock, releases the resource invariant when it calls unlock. It can do this, because it obtained the appropriate handle and full resource invariant *i.e.*, handle<T,L,f>, in the lock method. Since also the appropriate shares of the invariant are available, the precondition of sync.set(U) holds. The postcondition provides a handle on U, and empty resource.

6.3 SingleCell

Finally, we prove that the SingleCell example does not have data race, which is essential for our motivating lock-less hash table. We define the required functions and predicates as follows:

```
group inv<frac p> = Perm(data,p);
pred trans<role r,int c,int n> =
     (c==E && n==W) || (c==W && n==D);
frac  share(role r,int s) {
     if(r == S && s == E) return 1;
     if((r == T || r==S) && s == D) return +0;
     return 0;}
```

In the definition of share we use +0 to indicate the ϵ fraction. Lst. 8 contains the full proof outline.

Verification of the constructor is exactly as for SpinLock: we provide the synchronizer with the full resource invariant, and obtain a handle for role T initialized with E.

Method find_or_put is verified as follows. Any thread calling this method holds a handle for E. The precondition of sync.compareAndSet(E,W) requires inv<share(S,W)>, inv<share(T,W)> and trans<T,E,W>, which all are true. If the call is successful, the thread obtains inv<share(S,E)>, which can be opened to obtain permission to write data. In this case, next sync.set(D) will be called. The permission to access data can be closed to obtain inv<share(S,D)>. After the call, this thread only holds inv<+0>, and knows that data == v. It will return with value PUT and the method's postcondition is established. If the call to compareAndSet was not successful, the thread invokes sync.get(). This only requires a handle on E, since share(T,E) = 0. When the get method returns D, the thread obtains inv<+0>, and thus it can read data. If data == v, the method returns with SEEN and the postcondition of the method is established. Otherwise, the method returns COLN and the postcondition trivially holds.

```
public class SingleCell{
  ... // fields, invariants, predicates and protocols
  //@ requires inv<1>;
  //@ ensures handle<T,E,1>;
  SingleCell(){
   {inv<share(S,E)>}
   sync = new AtomicInteger(E);
   {handle<T,E,1>}
  }
  //@ requires handle<T,E,f>;
  //@ ensures handle<T,D,f>;
  //@ ensures \result == PUT ==> PointsTo(data,+0,v);
  //@ ensures \result == SEEN ==> PointsTo(data,+0,v);
  int find_or_put(int v){
{handle<T,E,f>*trans<T,E,W>*inv<share(T,E)>*inv<share(S,W)>}

    if(sync.compareAndSet(E,W)){
{handle<T,W,f>*inv<share(T,W)>*inv<share(S,E)>}
{handle<T,W,f>*Perm(data,1)}
      data = v;
{handle<T,W,f>*PointsTo(data,1,v)}
{handle<T,W,f>*trans<T,W,D>*inv<share(T,W)>*inv<share(S,D)>}

      sync.set(D);
{handle<T,D,f>*inv<share(T,D)>*inv<share(S,D)>*(data==v)}

{handle<T,D,f>*Perm(data,+0)*(data==v)}
      {handle<T,D,f>*PointsTo(data,+0,v)}
      return PUT;
    }
{handle<T,E,f>*inv<share(T,E)>*inv<share(S,W)>}
    if(sync.get()!=E){
{handle<T,val,f>*inv<share(T,val)>*(val!=E)}
      while(sync.get()==W);
{handle<T,val,f>*inv<share(T,val)>*(val!=E)*(val!=W)}
      if(sync.get() == D)
{handle<T,D,f>*inv<share(T,D)>}
{handle<T,D,f>*Perm(data,+0)}
       if(data == v)
         {handle<T,D,f>*PointsTo(data,+0,v)}
         return SEEN;
         {handle<T,D,f>*PointsTo(data,+0,val)*(val!=v)}
       else
         return COLN;
    }
  }
}
```

Lst. 8. Annotated SingleCell storage

7. Related Work

Reasoning about atomic operations and in particular non-blocking algorithms using Separation Logic is an active research area. Vafeiadis and Parkinson combined Rely-Guarantee reasoning and Separation Logic in RGSep to reason about concurrent programs [22]. Assertions in RGSep distinguish between local and shared state. Actions are used to describe the interferences on the shared state between parallel processes. Bornat and Amjad [3] used this logic to reason about non-blocking inter-process buffers. Young *et al.* embedded permission-annotated actions in their assertion language and extend abstract predicates to Concurrent Abstract Predicates (CAP) [6]. In our approach we tried to encode the permissions on actions in the usage protocols and contracts. Authorizing the winner thread with the full fraction held by the synchronizer, transfers the full permission on doing atomic write action,*i.e.*, set, to the winner thread. However, using CAP one can not encode the usage protocol, which is the crucial ingredient of our approach. Recently, Svendsen *et al.* presented a higher-order separation logic with CAP [21], which supports client usage protocols. We will investigate how this precisely relates to our approach. Parkinson *et al.* proved absence of the ABA problem in a non-blocking stack using permission-based separation logic [16]. This algorithm is a typical example of the GC pattern using AtomicReference. We need to adapt our specifications to AtomicReference to be able to reason about GC patterns. Jacobs [9] and Smans are working on modular verification of atomic operations in non-blocking algorithms. So far, only preliminary results have been published. Finally, Sutherland *et al.* [20] presented a different approach. They used thread *colouring* to express the thread's role and annotate thread usage policies. Our idea to identify roles is inspired by the notion of thread colour. However, Sutherland *et al.* neither consider the lack of the data races nor the correctness of synchronization mechanisms.

8. Conclusion

This paper discusses different patterns that are used to define a synchronization mechanism for *exclusive* access of a shared resource using the basic atomic operations *read*, *write* and *conditional write*. Based on these patterns, we provide a specification of the class AtomicInteger from the java.util.concurrent API, using permission-based Separation Logic. The specification is parametrised by: the thread's roles; a resource invariant, describing the shared resource that is protected

by the synchronizer; a relation defining allowed state changes; and a function that describes for each state change which share of the resource invariant is transferred from the thread to the synchronizer, or vice versa. Preconditions of the atomic operation prescribe which share of the resource invariant is transferred into the synchronizer, while postconditions specify which share is transferred from the synchronizer to the thread.

For each of the synchronization patterns we showed how the specification can be instantiated, so that a typical example implementation of this pattern can be verified. To ensure overall soundness of the approach, it has to be ensured that the sharing function does not implicitly allow the creation of resources. We also discussed how this can be verified.

As future work, we plan to extend our approach to synchronizers for shared usage. This will allow us to verify reference implementations of shared usage classes such as `Semaphore`, `ReadWriteReentrantLock` and `CountDownLatch`. Specification of these synchronization mechanisms are already proposed in [1]. As mentioned, the `SingleCell` example presented here is a simplified version of the lock-less hash table used in the LTSmin model checker. As future work, we will also specify and verify a complete Java implementation of this hash table, where each table cell is protected by an element in the array along with storing the hash key of the data. This requires to leverage our specification of `AtomicInteger` to `AtomicLongArray`, and to have support for reasoning about arrays. Finally, leveraging the specification also to `AtomicReference` will allow us to reason about typical lock-free data structures such as Java's `ConcurrentLinkedQueue`.

Acknowledgments

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

References

[1] A. Amighi, S. Blom, M. Huisman, W. Mostowski, and M. Zaharieva-Stojanovski. Synchronisation primitives in separation logic, 201x. Submitted.

[2] S. Blom, J. van de Pol, and M. Weber. LTSmin: Distributed and symbolic reachability. In T. Touili, B. Cook, and P. Jackson, editors, *CAV*, volume 6174 of *LNCS*, pages 354–359. Springer, 2010.

[3] R. Bornat and H. Amjad. Inter-process buffers in separation logic with rely-guarantee. *Formal Asp. Comput.*, 22(6):735–772, 2010.

[4] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In J. Palsberg and M. Abadi, editors, *POPL*, pages 259–270. ACM, 2005.

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

[6] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In T. D'Hondt, editor, *ECOOP*, volume 6183 of *LNCS*, pages 504–528. Springer, 2010.

[7] C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java's reentrant locks. In G. Ramalingam, editor, *Asian Programming Languages and Systems Symposium*, volume 5356 of *LNCS*, pages 171–187. Springer, 2008.

[8] C. Haack, M. Huisman, C. Hurlin, and A. Amighi. Permission-based separation logic for multithreaded Java programs, 201x. Submitted.

[9] B. Jacobs and F. Piessens. Modular full functional specification and verification of lock-free data structures. CW Reports CW551, Department of Computer Science, K.U.Leuven, 2009.

[10] A. Laarman, J. van de Pol, and M. Weber. Boosting multi-core reachability performance with shared hash tables. In R. Bloem and N. Sharygina, editors, *FMCAD*, pages 247–255. IEEE, 2010.

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

[12] J. Manson, W. Pugh, and S. Adve. The Java memory model. In *Principles of Programming Languages*, pages 378–391, 2005.

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

[14] M. Parkinson and G. Bierman. Separation logic, abstraction and inheritance. In *Principles of programming languages (POPL '08)*, pages 75–86. ACM Press, 2008.

[15] M. Parkinson and A. Summers. The relationship between separation logic and implicit dynamic frames. In G. Barthe, editor, *ESOP*, volume 6602 of *LNCS*, pages 439–458. Springer, 2011.

[16] M. Parkinson, R. Bornat, and P. O'Hearn. Modular verification of a non-blocking stack. In M. Hofmann and M. Felleisen, editors, *POPL*, pages 297–302. ACM, 2007.

[17] M. J. Parkinson. Local reasoning for Java. Technical Report UCAM-CL-TR-654, University of Cambridge, Computer Laboratory, Nov. 2005.

[18] M. Raynal. *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer, 2013.

[19] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.

[20] D. F. Sutherland and W. L. Scherlis. Composable thread coloring. In *Principles and Practice of Parallel Programming*, PPoPP '10, pages 233–244. ACM, 2010.

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

[22] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In L. Caires and V. T. Vasconcelos, editors, *CONCUR*, volume 4703 of *LNCS*, pages 256–271. Springer, 2007.