Resource Protection Using Atomics
Patterns and Verification

Afshin Amighi, Stefan Blom, and Marieke Huisman

Formal Methods and Tools, University of Twente, Enschede, The Netherlands
{a.amighi,s.blom,m.huisman}@utwente.nl

Abstract. For the verification of concurrent programs, it is essential to be able to show that synchronisation mechanisms are implemented correctly. A common way to implement such sychronisers is by using atomic operations. This paper identifies what different synchronisation patterns can be implemented by using atomic read, write and compare-and-set operation. Additionally, this paper proposes also a specification of these operations in Java's `AtomicInteger` class, and shows how different synchronisation mechanisms can be built and verified using atomic integer as the synchronisation primitive.

The specifications for the methods in the `AtomicInteger` class are derived from the classical concurrent separation logic rule for atomic operations. A main characteristic of our specification is its ease of use. To verify an implementation of a synchronisation mechanism, the user only has to specify (1) what are the different roles of the threads participating in the synchronisation, (2) what are the legal state transitions in the synchroniser, and (3) what share of the resource invariant can be obtained in a certain state, given the role of the current thread. The approach is illustrated on several synchronisation mechanisms. For all implementations, we provide a machine-checked proof that the implementations correctly implement the synchroniser.

1 Introduction

Motivation To increase performance, multi-threaded applications should make optimal use of multi-core architectures. Typically, these applications exploit synchronisation to ensure that there are no conflicting accesses to shared resources. In shared-memory concurrency, atomic variables, *i.e.* variables that may only be accessed using atomic operations, are used to implement these synchronisation mechanisms. Such variables are called *atomic synchronisers*. In programming languages like Java, atomic variables along with three basic atomic operations (atomic read, write and compare-and-set) are encapsulated in *atomic classes*. To guarantee the correctness of concurrent programs it is essential to be able to reason about these atomic classes. This paper proposes an approach to specify the behavior of an *atomic class* as a synchroniser.

Approach. We provide specifications for the basic atomic operations, read, write and conditional update, which can be used to verify the implementation of different synchronisation classes. These specifications are derived from the general

J. Garrigue (Ed.): APLAS 2014, LNCS 8858, pp. 255–274, 2014.

rule for atomic operations from a well-established program logic, Concurrent Separation Logic (CSL) [23]. The specifications have been designed to be easy to use: when using them to show the correctness of a concrete synchroniser implementation, only a few intuitive parameters have to be provided. This paper, presents a specification for atomic integers, as it is the base for most of the synchronisation classes in Java. However, our approach also works for the other atomic classes.

In our approach, any thread has a local view of the atomic variable. The global state is then defined in terms of the atomic variable and all the local views. In addition, the atomic variable is instrumented with a protocol that describes what the legal state transitions are. The protocol is used by the thread to derive the guarantees that the environment provides. Additionally, a resource invariant is declared, which specifies which resources are protected by the synchroniser. The derived specifications for the `AtomicInteger` operations are thus parametrised by this protocol and the resource invariant. This specification expresses how `AtomicInteger`, as an atomic synchroniser, grants and retains permissions to access the shared resource specified by the resource invariant *exclusively*. To describe the specifications and the predicates encoding the views and the protocol, we use permission-based separation logic for Java [3,9].

Before presenting the specification of `AtomicInteger`, we introduce several synchronisation patterns, each supported by an example. For each of these synchronisation patterns, we discuss how the specification parameters 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.

Contributions. The main contributions of this paper are the following: (1) an overview of typical synchronisation patterns using the basic atomic operations; (2) a general specification for the basic atomic operations that together can synchronise a group of threads; (3) a simple, practical and thread-modular contract for `AtomicInteger` as a synchroniser; and (4) verification of several examples implementing the synchronisation patterns using our VerCors tool set [25].

Outline. This paper is structured as follows: in Section 2 we present the different synchronisation patterns using `AtomicInteger` as a synchronisation primitive. Section 3 derives contracts for atomic read, write, and compare-and-set. Section 4 explains the generalised specification of the `AtomicInteger` class and discusses correctness proofs of the clients using `AtomicInteger`. Finally, Section 5 presents related work and Section 6 draws conclusions, and discusses future work.

2 Synchronisation in Java

To support thread-safe access to single variables, Java provides the package `java.util.concurrent.atomic`, as part of Java's general concurrency API. This package provides wrappers for `volatile` variables with appropriate atomic operations for read, write, and compare-and-swap. In Java, changes to a volatile

variable are immediately visible to other threads, *i.e.* their value will never be cached thread-locally. This makes volatile variables suitable to implement synchronisation mechanisms, where it is essential that all threads have a consistent view of the synchroniser.

This paper particularly studies the `AtomicInteger` class, which encapsulates a volatile field of type integer. Essentially, it provides the following methods: `get()`, returning the value that was last written to the field; `set(int v)`, atomically assigning the value `v` to the 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 returning a boolean to indicate whether the update succeeded.

Synchronisation Patterns. In a shared-memory concurrency setting, two kinds of thread interactions via a synchronizer can be distinguished: *cooperation* and *competition* [18]. In a cooperative interaction, threads employ a *cooperative synchroniser* as a communication channel to cooperatively share a resource. In a competitive interaction, a *competitive synchroniser* runs a competition and provides (temporary) access to the shared resource to the winner. A synchroniser can behave cooperatively or competitively in different states, this is called a *hybrid* interaction. Various patterns of synchronisation can be described in terms of atomic integer operations:

GS (`get` and `set`). Threads can cooperatively interact using atomic read and write. Every thread has a designated state in which it obtains the resource, and all threads attempt to reach their designated state. When a thread writes to the atomic integer, it implicitly signals who *should* own the resource next (cooperation). Based on the value written into the synchroniser, ownership of the resource is transferred to the appropriate thread waiting for that particular value. Producer-Consumer and Dekker's critical section algorithm are examples of this pattern. Lst. 1 shows `ProducerConsumer` with two methods `produce` and `consume`, sharing a field `data`, that implements this algorithm. Typically, these methods will be executed as part of a surrounding loop. The `AtomicInteger` denotes the state of the buffer: full (`F`) or empty (`E`). Both the producer and the consumer wait until the buffer gets into their desired 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.

SC (`set` and `compareAndSet`). Atomic write and conditional update can be used to implement a competitive synchroniser. Threads are competing to obtain the protected resource by calling `compareAndSet`. A thread that succeeds in changing the state, obtains the resource. When it no longer needs the resource, it sets the state to the initial value, to signal its availability. Failing threads continue to try to acquire the resource by checking whether the state is reverted back to the initial state. A spin-lock implementation using `AtomicInteger` (see Lst. 2) is a known example of this pattern where the atomic integer value encapsulates the state of the lock: locked (`L`) or unlocked (`U`). If a thread successfully updates the state from `U` to `L`, it acquires

```
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();
   sync.set(F); // signal
   while(sync.get() == F); // wait
 }
 void consume(){
   while(sync.get()==E);  // wait
   read();
   sync.set(E); // signal
 }
 // methods write() and read()
}
```

Lst. 1. `ProducerConsumer`: cooperation

```
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`: competition

the lock (method `lock`). Consequently, failing threads enter a try-wait loop, until the lock is released. To release the lock, the thread holding the lock executes `set(U)` (method `unlock`).

GC (`get` and `compareAndSet`). Atomic read and conditional update are suited to implement a synchronisation mechanism that *partially* transfers the resources between the participating threads. Shared reading synchronisation mechanisms using `AtomicInteger` like `Semaphore` and `CountDownLatch` are typical instances of this pattern. Also lock-free pointer-based data structures using `AtomicReference` are examples of this pattern. Since, here, we are only looking at exclusive synchronisation mechanisms, we do not discuss this pattern further. However, a generalisation of our approach to reason about partial resource ownership using atomics is ongoing work.

GSC (`get`, `set` and `compareAndSet`). All basic operations of `AtomicInteger` can be used together to implement a hybrid synchroniser. Threads compete with each other to obtain the resource by calling `compareAndSet`. A thread that succeeds in changing the state, wins the resource. Failing threads may not compete any more to change the state. But, they have to wait for the resource availability. When the winner thread no longer needs the resource, it updates the state to signal how the resource should be used afterwards. Lst. 3 shows the implementation of a `SingleCell` algorithm, which illustrates a hybrid pattern[1]. It provides a single method to find or put a value in a shared storage cell. The storage cell is always in one of these states: empty (`E`), writing (`W`) or done (`D`). The cell containing the value (the state `D`) must be immutable. Initially, all threads are competing to assign their value. If a thread succeeds in obtaining writing access to the resource, the state becomes `W`. After completing the assignment, it will report its success (returns `PUT`). All other threads have to wait until the value is assigned, and then they

[1] This is a simplified version of a lock-less hash table, especially designed for state space exploration in the multi-core model checker LTSmin [12].

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

Lst. 3. `SingleCell`: hybrid

check 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 a collision (returns `COLN`).

3 Ownership Exchange via Atomics

This section derives permission-based Separation Logic contracts for atomic read, write, and compare-and-set operations. Separation Logic (SL) is an extension of Hoare Logic, originally developed to reason about programs with pointers [19]. A key characteristic of SL is that it allows to reason about disjointness of heaps. This ability also makes SL suitable to reason in a thread-modular way about multi-threaded programs, as demonstrated by O'Hearn [15], who introduced Concurrent Separation Logic (CSL) to verify correct access of synchronised threads to dynamically allocated memory. CSL also introduced the notion of *ownership*, to specify how a synchronisation construct *exclusively* exchanges ownership of a memory location. To be able to verify programs where multiple threads concurrently read a shared address, permission-based separation logic [3] extends CSL with fractional permissions [4]. In a fractional permission model, a thread holds a permission $\pi \in (0, 1]$ to access a heap location. Full ownership, providing write permission, is indicated by the full permission 1, while any permission $\pi \in (0, 1)$ indicates a read-only access.

Let E denote arithmetic expressions, B boolean expressions and R pure resource formulas. In our fragment of CSL, the syntax for assertions P is defined as follows:

$$
\begin{array}{lll}
B & ::= & \neg B \mid B_1 \wedge B_2 \mid B_1 \vee B_2 \\
R & ::= & \mathsf{emp} \mid E_1 \overset{\pi}{\mapsto} E_2 \mid R_1 * R_2 \mid R_1 \mathrel{-\!\!*} R_2 \\
P & ::= & B \mid R \mid B * R \mid B \implies R \mid \forall x.\ P \mid \exists x.\ P \mid \circledast_{i \in \mathsf{I}}\ P_i
\end{array}
$$

In addition to the classical connectives and first order quantifiers, the main assertions are: (1) the empty heap assertion, written emp, (2) the *points-to* predicate $E_1 \overset{\pi}{\mapsto} E_2$, meaning that expresssion E_1 points to a location on the heap, has permission π to access this location, and this location contains the value E_2, (3) the *separating conjunction* operator $*$, expressing that two formulas are valid for disjoint parts of the heap, (4) a magic wand (also known as resource implication) formula $R_1 \mathrel{-\!\!*} R_2$ which holds for any heap that has the following property: if the heap is extended with a *disjoint* heap that satisfies R_1, then the combined heap satisfies R_2, and finally (5) an iterative separating conjunction over a set I, written $\circledast_{i \in \mathsf{I}} P_i$. Below, we use $[E]$ to denote the contents of the heap at location E and we use $E \mapsto -$ to indicate that the content stored at location $[E]$ is not important.

Permissions can be transferred between threads at synchronisation points (including thread creation and joining). Moreover, permissions can be split and combined to change between read and write permissions:

$$E_1 \overset{\pi}{\mapsto} E_2 * E_1 \overset{\pi'}{\mapsto} E_2 \Leftrightarrow E_1 \overset{\pi+\pi'}{\mapsto} E_2$$

The addition of two permissions is undefined if the result is greater than the full permission. 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.

3.1 Basic Rules

Next we show how the contracts in permission-based SL for the basic atomic operations can be derived. We base ourselves on the work by Vafeiadis [23], which enables us to define a language where atomic commands, denoted $\langle C \rangle$, are the only constructs for synchronisation.

We divide the domain of the heap into a set of *atomic* locations ALoc (*e.g.*, the volatile field of `AtomicInteger`) and a set of *non-atomic* locations NLoc (*e.g.*, `data` in Lst. 1). An atomic location $s \in \mathsf{ALoc}$ may only be accessed using: (1) $\texttt{get}(s)$ for atomic read of the atomic location s, (2) $\texttt{set}(s, n)$ for atomic update of s with n, and (3) $\texttt{cas}(s, x, n)$ for atomic conditional update of s. We use the term *atomic value* to refer to the value that an atomic variable contains and the term *resources* to refer to *non-atomic* locations of the heap.

As proposed by O'Hearn, in a concurrent setting a resource invariant is attached with a synchroniser. This associates ownership of a part of the state space with possible states of the synchroniser [15]. For example, the resource invariant for a lock $\mathsf{lock} \in \mathsf{ALoc}$ that protects the resource $\mathsf{x} \in \mathsf{NLoc}$ is defined as:

$$I_{\mathsf{lock}} = \exists\, v \in \{0,1\}.\ \mathsf{lock} \overset{1}{\mapsto} v * ((v = 1 \implies \mathsf{emp}) * (v = 0 \implies \mathsf{x} \overset{1}{\mapsto} -))$$

This expresses that full ownership of the location x is available to win when $[\mathsf{lock}] = 0$, while if $[\mathsf{lock}] = 1$ then emp (interpreted as nothing) can be obtained.

In general, using a function res that maps an atomic value to a set of disjoint resources, given Val as the set of values and $s \in \mathsf{ALoc}$, the resource invariant I_s is defined as:

$$I_s = \exists\, v \in \mathsf{Val}.\ s \overset{1}{\mapsto} v * \mathsf{S}(s,v) \quad \text{where} \quad \mathsf{S}(s,v) = \circledast_{r \in \mathsf{res}(s,v)} r \overset{1}{\mapsto} -$$

In CSL, a judgment $I \vdash \{P\}\ C\ \{Q\}$ expresses the following: given a globally accessible resource invariant I and a local precondition P, if a statement C starts its execution in a state satisfying $P * I$, and if C terminates, then its final state satisfies $Q * I$. The proof rule for atomic commands [23] expresses that to prove correctness of $\langle C \rangle$, the resource invariant I can be used for the verification of the atomic body C. Thus, I is not accessible to the environment. Moreover, within the body C, the resource invariant I may be invalidated, because it is not visible to the environment, but it must be re-established before C is finished:

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

We use the rule [Atomic] to derive specifications for the basic atomic operations `get`, `set` and `cas` when they are coordinating a set of threads to (exclusively) access a shared resource. The specifications should capture all exclusive synchronisation patterns mentioned above: cooperative, competitive and hybrid. Therefore, we need to enrich the resource invariant definition with an abstraction of local state and feasible states, which allows one to deduce what the environment guarantees. Next,we instantiate the [Atomic] rule to derive the resources that `set`, `get` and `cas` exchange to perform exclusive access synchronisation.

3.2 Synchronisation Protocol

Assuming a set of threads Thr, for each atomic location s that is synchronising the threads, we define the *view* of a thread $t \in \mathsf{Thr}$ as an *atomic ghost variable*, denoted s_t. Each thread stores the last visited atomic value in its view. We define the view to be atomic in order to restrict the thread t, using ghost code, to update *its* view *only* inside an atomic block. To do so, the ownership of a view is split in half between the owner thread and the resource invariant, *i.e.* the shared state. Therefore, a thread can always read its own view, but it can only update its view when it captures the other half permission inside an atomic block by accessing the resource invariant. Views of threads indexed by thread identifiers are written as a vector of views $\vec{s_t}$. Similarly, $\vec{v_t}$ denotes a vector of values pointed to by the views, indexed by the corresponding thread identifiers, while $\vec{v_{t\{v_\tau = x\}}}$ denotes a vector such that the item indexed with τ is equal to x. For the sake of simplicity we assume that there is only one single atomic location s functioning as the synchroniser. However, the approach is generalisable for multiple atomic location.

We define the *(global) atomic state* as a tuple of the atomic value and all thread local views of it, denoted $(s, \vec{s_t})$. An atomic state is *admissible* if at least

one thread has a correct view of the synchroniser. An admissible atomic state is *feasible* if either (1) it is an initialisation state where all the threads have an identical view of the initialised atomic location, or (2) it is reachable from the initialisation state by a finite set of atomic operations.

As the views must be updated only inside the atomic operations, they can reflect the actions that the environment can perform w.r.t. the atomic location. The current definition of the resource invariant is too restrictive to reflect this. So, first, we define the protocol of the synchroniser in terms of the atomic state:

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

where fsbl determines whether the *atomic state* is feasible.

Example 1 (Protocol for `ProducerConsumer`*).* To illustrate our definition of feasible states, consider the `ProducerConsumer` example, where we have two threads p (producer) and c (consumer) with corresponding views, *i.e.* $\mathsf{s_p}$ and $\mathsf{s_c}$, respectively, given an atomic variable s:

$$\begin{aligned}\mathsf{P}_{\mathsf{s}}^{\{\mathsf{p},\mathsf{c}\}} = (\ &([\mathsf{s}] = E \wedge [\mathsf{s_p}] = E \wedge [\mathsf{s_c}] = E) \vee ([\mathsf{s}] = F \wedge [\mathsf{s_p}] = F \wedge [\mathsf{s_c}] = E) \vee \\ &([\mathsf{s}] = F \wedge [\mathsf{s_p}] = F \wedge [\mathsf{s_c}] = F) \vee ([\mathsf{s}] = E \wedge [\mathsf{s_p}] = F \wedge [\mathsf{s_c}] = E)\ )\end{aligned}$$

Note that $([\mathsf{s}] = F, [\mathsf{s_p}] = E, [\mathsf{s_c}] = F)$ is not a feasible state. Therefore, when p believes that the buffer is empty (E), it can safely *rely* on the fact that no other thread is allowed to modify s to full (F). Thus, p deduces that it exclusively owns s, so $[\mathsf{s}]$ *must be* E when $[\mathsf{s_p}] = E$.

Example 2 (Protocol for `SpinLock`*).* Consider the `SpinLock` example, which is a competititve pattern. Its protocol is defined as follows:

$$\begin{aligned}\mathsf{P}_{\mathsf{s}}^{\mathsf{Thr}} = ([\mathsf{s}] = U\ \wedge (\forall\ t \in \mathsf{Thr}.\ [\mathsf{s}_t] = U)) \vee \\ ([\mathsf{s}] = L\ \wedge (\exists\ \tau \in \mathsf{Thr}.\ [\mathsf{s}_\tau] = L\ \wedge \forall\ t \in \mathsf{Thr} \setminus \{\tau\}.\ [\mathsf{s}_t] = U))\end{aligned}$$

This expresses that either the lock is available and all threads have a correct view of the state, or there is only one thread that has acquired the lock and updated its view while all others have failed to change their beliefs. This makes it possible for the unlocking thread to rely on its view, knowing that it will be the only one that has the correct view.

The protocol suffices to derive the contracts for the basic atomic operations when they are involved in a *competitive* pattern. To cover cooperative patterns, where *threads obtain the shared resources based on their views*, in addition, the resource invariant has to express what resources are protected in terms of the atomic state. In fact, instead of one single atomic variable s, $(s, \vec{s_t})$ plays the role of a global synchroniser. Similar to res, we define ares to map the atomic state to a set of disjoint resources. Therefore, we replace $\mathsf{S}(s, v)$ with $\mathsf{R}(s, v, \vec{s_t}, \vec{w_t})$ to denote all the resources associated with $[s] = v$ and $[\vec{s_t}] = \vec{w_t}$.

Now we are ready to define precisely what we mean by a synchronisation primitive, based on our extended definition of resource invariant.

Definition 1 (State-based Synchroniser). *An atomic location s together with the basic atomic operations* $\mathsf{ACmd} = \{\mathtt{get}, \mathtt{set}, \mathtt{cas}\}$ *define a* state-based primitive synchronisation mechanism *for a set of threads* Thr *if it is instrumented with a resource invariant defined as follows:*

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

where $R(s, v, \vec{s_t}, \vec{w_t}) = \circledast_{r \in \mathit{ares}(s, v, \vec{s_t}, \vec{w_t})} r \overset{1}{\mapsto} -$.

Example 3 (Synchroniser for `ProducerConsumer`*).* Based on the protocol defined in Example 1, we define the resource invariant of the atomic synchroniser s to synchronise p and c:

$$I_{\mathsf{s}} = \exists v, w_{\mathsf{p}}, w_{\mathsf{c}} \in \{E, F\} \cdot \mathsf{s} \overset{1}{\mapsto} v * \mathsf{s_p} \overset{\frac{1}{2}}{\mapsto} w_{\mathsf{p}} * \mathsf{s_c} \overset{\frac{1}{2}}{\mapsto} w_{\mathsf{c}} * \mathsf{R}(\mathsf{s}, v, \vec{s_t}, \vec{w_t}) * \mathsf{P}_{\mathsf{s}}^{\{\mathsf{p},\mathsf{c}\}}$$

where $\mathsf{R}(\mathsf{s}, v, \vec{s_t}, \vec{w_t})$ is $\mathtt{data} \overset{1}{\mapsto} -$ if $v = E$, $w_{\mathsf{p}} = F$, $w_{\mathsf{c}} = E$ and $v = F$, $w_{\mathsf{p}} = F$, $w_{\mathsf{c}} = E$, and $\mathsf{R}(\mathsf{s}, v, \vec{s_t}, \vec{w_t})$ is emp if threads agree on the value of s. This expresses that s holds the full ownership of `data` when threads do not agree on the value of the synchroniser (*i.e.*, during the transition phase).

Example 4 (Synchroniser for `SpinLock`*).* Considering the `SpinLock` protocol in Example 2, we define the resource invariant for s. Here, regardless of the views of the threads, the resource invariant holds the full resource when the state is U, otherwise the winning thread holds it.

$$I_{\mathsf{s}} = \exists v, \vec{w_t} \in \{U, L\} \cdot \mathsf{s} \overset{1}{\mapsto} v * (\circledast_{t \in \mathsf{Thr}} s_t \overset{\frac{1}{2}}{\mapsto} w_t) * \mathsf{R}(\mathsf{s}, v, \vec{s_t}, \vec{w_t}) * \mathsf{P}_{\mathsf{s}}^{\mathsf{Thr}}$$

where $\mathsf{R}(\mathsf{s}, v, \vec{s_t}, \vec{w_t})$ will be $\mathtt{data} \overset{1}{\mapsto} -$ when $v = U$ and emp when $v = L$.

Next we investigate how the three basic atomic operations can exchange the shared resources.

3.3 Specifications of Atomics

This section derives contracts for the three basic atomic operations for state-based synchronisation. The contracts, shown in Figure 1, essentially express that in an exclusive state-based synchronisation, the thread τ executing an atomic operation to update the state of the synchroniser, should *provide* the resources associated with the state after the operation, and in return will *receive* the resources associated with the previous state of the synchroniser. In Figure 1, we used $R_s^{\mathsf{Thr}}(\tau, x, y)$ to denote all the resources when $s = x$ and $s_\tau = y$.

Our technical report [1] presents the complete derivations. Basically, for each basic atomic operation we propose an implementation using basic instructions. We instantiate [Atomic] for each operation with a precondition about the

$$Let\ R_s^{\mathsf{Thr}}(\tau, x, y) = \circledast_{\vec{v_t}\{v_\tau = y\} \in \mathsf{Val}.\, \mathsf{fsbl}(x, \vec{v_t}\{v_\tau = y\})} \mathsf{R}(s, x, \vec{s_t}, \vec{v_t}\{v_\tau = y\})$$

$$\frac{\forall\ v, \vec{v_t} \in \mathsf{Val}.\ v_\tau = \mathsf{d} \wedge \mathsf{fsbl}(v, \vec{v_t}\{v_\tau = \mathsf{d}\}) \implies \mathsf{fsbl}(n, \vec{v_t}\{v_\tau = n\})}{I_s \vdash \{s_\tau \overset{\frac{1}{2}}{\mapsto} \mathsf{d} * R_s^{\mathsf{Thr}}(\tau, n, n)\}\ \mathtt{set}_\tau(s, n)\ \{s_\tau \overset{\frac{1}{2}}{\mapsto} n * R_s^{\mathsf{Thr}}(\tau, \mathsf{d}, \mathsf{d})\}}\ [\textsc{WAtm}]$$

$$\frac{}{I_s \vdash \{s_\tau \overset{\frac{1}{2}}{\mapsto} \mathsf{d}\}\ \mathtt{get}_\tau(s)\ \{s_\tau \overset{\frac{1}{2}}{\mapsto} ret * (R_s^{\mathsf{Thr}}(\tau, ret, ret) \mathrel{-\!\!*} R_s^{\mathsf{Thr}}(\tau, ret, \mathsf{d}))\}}\ [\textsc{RAtm}]$$

$$\frac{\forall\ v, \vec{v_t} \in \mathsf{Val}.\ v_\tau = x \wedge \mathsf{fsbl}(v, \vec{v_t}\{v_\tau = x\}) \implies \mathsf{fsbl}(n, \vec{v_t}\{v_\tau = n\})}{I_s \vdash \{s_\tau \overset{\frac{1}{2}}{\mapsto} x * R_s^{\mathsf{Thr}}(\tau, n, n)\}\ \mathtt{cas}_\tau(s, x, n)\ \{(ret = \mathtt{true} \wedge s_\tau \overset{\frac{1}{2}}{\mapsto} n * R_s^{\mathsf{Thr}}(\tau, x, x)) \vee (ret = \mathtt{false} \wedge s_\tau \overset{\frac{1}{2}}{\mapsto} x * R_s^{\mathsf{Thr}}(\tau, n, n)\}}\ [\textsc{CAtm}]$$

Fig. 1. Contracts derived for `set`, `get` and `cas`

thread's view and thread's local state, containing the required resources. Then we derive the postcondition from the precondition and the body, taking into account that I_s is available inside the body, providing the resources associated to the current state of the synchroniser. Inside the body, either the atomic location or the view of the thread is updated. The derivations show that the thread consumes the resources it currently holds to re-establish I_s and exits the atomic body with an updated atomic state and the resources it obtains as the result of the update.

Atomic Write. Operation $\mathtt{set}_\tau(s, n)$ denotes the atomic update of s with n by a particular thread τ. We derive rule [WAtm], expressing that the executing thread with the view d delivers all the resources associated with the feasible atomic state after the update. We should stress here that this contract is specific to using atomic write for synchronisation, it is not the most general contract possible.

For an atomic synchroniser for *exclusive* resource access, it is crucial that the value inferred by the protocol coincides with the thread's view. In other word, the protocol embedded in the resource invariant must prove that the thread executing an atomic write has the *full permission* to do the `set` action, otherwise, it is not guaranteed that the thread intended to execute `set`, can indeed accomplish this safely.

Atomic read. The read action for a particular thread $\tau \in \mathsf{Thr}$ with a view s_τ that has the last visited value d from the atomic value s is indicated by $\mathtt{get}_\tau(s)$. In the rule [RAtm], the contract of the atomic read specifies that the atomic variable does not change its value, while the atomic state is modified because the reading thread updates its view. So the thread has to establish the resource invariant with the resources associated with the updated view inside the atomic body. As a result, it obtains the remainder as its postcondition, which is formalised using a magic wand operator. According to [19] this rule is correct if our resource assertions are *strictly exact*. In a fragment of CSL that we use as our specification language, all resource formulas are indeed strictly exact.

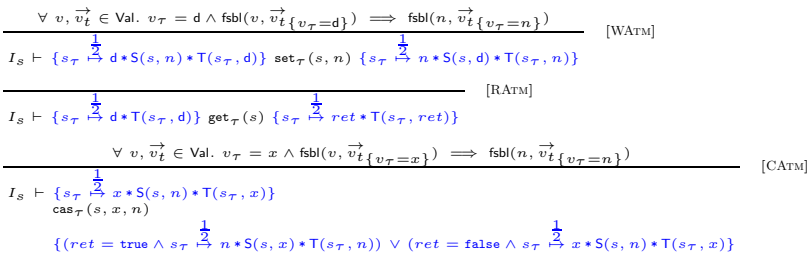

Fig. 2. Thread-modular specifications of atomic operations

Conditional update. Finally, rule [CAtm] specifies $\mathtt{cas}_\tau(s, x, n)$ with the expected value x and the value to be updated n. The calling thread assumes that the synchroniser contains a value equal to an expected value and then calls the operation to try to modify the atomic synchroniser to n. Therefore, the thread has to provide the resources associated with the updated atomic state and it will gain the resources associated with the expected value, if the operation succeeds. Otherwise, the operation returns all the provided resources.

3.4 Thread-Modular Contracts

The last step is to adapt the derived contracts for the atomic operations to a thread-modular specification. In particular, this means that the specifications should express the pre- and postconditions using local information only, *i.e.*, using (1) the atomic value as a globally known state, and (2) local information that contains the view of the executing thread.

Note that the resource invariant expresses when the *synchroniser* holds the resources. For example, the resource invariant of `ProducerConsumer` does *not* specify when a particular thread can obtain the buffer. Generally, in cooperative patterns, the synchroniser holds the resource *temporarily*, until one of the waiting threads updates its view. We take advantage of this to simplify the contracts by defining the resources using two components: (1) the resources that the *synchroniser* holds for the competition, which is used to associate resources to the atomic values in classical definition of the resource invariant, *i.e.* S, and (2) the resources that *threads* obtain when they are updating their views, denoted with T. Basically, $\mathsf{T}(s_\tau, v)$ indicates resources to be held by thread τ when $s_\tau = v$. We exploit these two components to decompose $R_s^{\mathsf{Thr}}(\tau, x, y)$ (defined in Figure 1) into a global and a thread local components.

These resources are either associated to the atomic value x, which will be obtained competitively using a `cas` operation, or associated to a particular view

of a thread, which will be obtained by updating the view. We can formally express this decomposition for $\tau \in \mathsf{Thr}$, $x, y \in \mathsf{Val}$ as:

$$R_s^{\mathsf{Thr}}(\tau, x, y) \Leftrightarrow \mathsf{S}(s, x) * \circledast_{t \in \mathsf{Thr}, v_t \in \mathsf{Val}} (\mathsf{T}(s_t, v_t) \mathrel{-\!\!*} \mathsf{T}(s_t, x))$$

where $\mathsf{T}(s_t, v_t) \mathrel{-\!\!*} \mathsf{T}(s_t, x)$ specifies the resources that thread t exchanges when it updates its view from v_t to x.

In summary, for a competitive pattern, resources are merely associated with the state of the synchroniser using $\mathsf{S}(s, x)$. A *cooperative pattern* exploits the definition of $\mathsf{T}(s_t, v_t)$, which associates the resources to the view of a thread expressing when the *thread* holds a resource. A *hybrid pattern* uses both $\mathsf{T}(s_t, v_t)$ and $\mathsf{S}(s, x)$ to reason about the resource exchanges.

We use this decomposition and update the contracts based on the fact that the executing thread may have resources obtained based on its *current* view. This results in thread-modular specifications for the basic atomic operations, as shown in Figure 2. which generally express that the executing thread must *provide* (1) the resources associated with its current view, and (2) the resources associated with the new state of the synchroniser. In return the thread *obtains* (1) the resources associated with its updated view, and (2) the resources associated with the previous state of the synchroniser. Note that in the patterns that we studied, the `cas` and `set` operations do not exchange resources using the thread views, and we are not aware of algorithms where these operations can transfer ownership based on their views.

4 Contracts of `AtomicInteger`

Based on the specifications derived above, we specify the behaviour of the `AtomicInteger` class as an *exclusive-access atomic synchronisation primitive*. First, we introduce our concrete specification language, which is a combination of permission-based SL and JML [5]. Then, we explain all predicates and functions that we use to specify `AtomicInteger`, and finally we present the complete specification.

4.1 Specification Language

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* [16], extended with permissions for concurrency.

In our assertion language we distinguish between *resource expressions* (R, typical elements r_i) and *functional expressions* (E, typical elements e_i), with the subset of logical expressions of type boolean (B, typical elements b_i). Formulas in our logic are defined by the following grammar:

$R ::= b \mid \mathtt{Perm}(e.\mathtt{f}, \mathsf{frac}) \mid r_1 \ ** \ r_2 \mid r_1 \ -\!\!* \ r_2$
$\quad \mid (\backslash\mathsf{forall} * \ \mathsf{T} \ \mathsf{v}; b; r) \mid b_1 ==\!> r_2 \mid e.\mathtt{P}(e_1, \cdots, e_n)$
$E ::=$ any *pure* expression $\quad B ::=$ any *pure* expression of type boolean

where T is an arbitrary type, v is a variable name, P is an abstract predicate [17] of a special type resource, f is a field name, and frac denotes a fractional permission.

The assertion Perm(e.f, π_i) expresses the access permission π_i of the field e.f. The notation for implication ==> is borrowed from JML. We also divert from the classical SL notation of * for the separating conjunction to ** in order to avoid name clashes with the multiplication operator. Given b as a constraint on the range of the quantifier we use \forall* to define the universal separating conjunction.

Assertions can also contain abstract predicates (P) that encapsulate the state space [17]. In our specification language e.P($e_1, \cdots, e_n$) expresses an invocation of the predicate P on the object e with arguments $e_1, \cdots, e_n$. Verifying a program, 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. Any predicate that is linear in its fractional arguments can be defined as a group. This means that the predicate can be split over permissions, see [8] for more details. When the value of a field is important we write PointsTo(x.f,p,v), which is equivalent to Perm(x.f,p) && x.f==v. Finally, we use the *minimum non-zero permission* [13], denoted as +0, to read an immutable location with the following axiom:

$$\texttt{Perm}(\texttt{x.f}, \texttt{+0}) ** \texttt{Perm}(\texttt{x.f}, \texttt{+0}) = \texttt{Perm}(\texttt{x.f}, \texttt{+0})$$

In addition, method and class specifications can be preceded by a given clause, declaring the method and class specification-only parameters. Method specification parameters are passed (implicitly) at method calls, class parameters are passed at type declaration and instance creation, resembling the parametric types mechanism of Java.

4.2 Predicates and Parameters

Any client program instantiating the AtomicInteger class as an exclusive atomic synchronisation primitive has to provide the *protocol* of the synchroniser object. In fact, a protocol of a synchronisation construct is an abstract state machine instrumented with an interpretation function that maps each state of the state machine to a fraction of the resources that the synchroniser object or a particular thread must hold in that state. Especially, in our settings, a protocol of a synchronisation construct must specify: (1) identification of the participants, (2) the shared resource that has to be protected by the synchronisation construct, (3) the fraction of the shared resource to be held by the synchroniser or a thread in each atomic state, and (4) the transitions that are valid for the synchroniser object.

To make a single specification of AtomicInteger that can capture all exclusive access patterns, the specification is parametrised by (1) a set of roles, which basically is an abstraction of the participating threads' identification, (2) an abstract predicate as a resource invariant, specifying the shared resources to

be protected by `AtomicInteger`, (3) a function to associate the states of the atomic integer as the synchroniser with the fraction of the shared resource, (4) a boolean predicate, encoding all the valid transitions that a particular instance of `AtomicInteger` can take, and (5) a handle token.

A *role* abstraction abstracts the identity of threads to a set of roles. This makes our specification unbounded in the number of threads. The synchroniser is defined as a globally known, special role, written `S`, that coordinates the threads. This role is declared as a publicly visible constant in class `AtomicInteger`, to hold the resource when the class runs the competition.

The validity of the transitions is encoded in the `trans` predicate. More importantly this encoding enables us to extract the set of the feasible states. The `trans` predicate expects as arguments the role of the invoking thread, the current and the intended update state of the synchroniser.

The shared resources are described by `inv(frac p)`, a resource formula parametrised with permissions (of type `frac`), and defined as a `group`, *i.e.* it should be splittable over permissions. To associate the *fraction of the shared resources* with the state of the atomic integer, we define the function `share`, which is parametrised by a role, and the value of the atomic integer. Our role abstraction allows us to express S and T in the specification presented in Figure 2 using only `inv` parametrised with `share`.

For example, instantiating `AtomicInteger` for `ProducerConsumer` we define:

```
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){
  return (r == P && s == E) ? 1: ((r == C && s == F) ? 1: 0); }
```

where the definition of `share` shows that the full ownership of the shared resource, *i.e.* `data`, is only associated with the views of the threads. In the specification presented in Figure 2 this would mean that the S component would be `emp` and the T component associates the full ownership of `data` to the views of the threads. Similarly, instantiating `AtomicInteger` for `SpinLock` we use these definitions:

```
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;  };
```

where `resinv` would be the shared resource to be protected by the lock which is passed as a class parameter to `SpinLock`. As it is specified in the definition of `share` the synchroniser, defined with the globally known role `S`, will hold the full resource when its state is `U` (unlocked). This can be expressed in the specification presented in Figure 2 with T defined as `emp` while the component S associates the full ownership of `resinv` to the unlocked state of the atomic location.

To invoke an operation from `AtomicInteger`, the calling thread must provide the correct required arguments which are demanded by the contracts. 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.

```
//@ given group (frac->group) inv;
//@ given (role,int->frac) share;
//@ given (role,int,int-> boolean) trans;
//@ given Set<role> rs;
class AtomicInteger {
  private volatile int value;
  //@ group handle(role r,int d,frac p);

  /*@
  requires inv(share(S,v));
  ensures (\forall* role r; rs.contains(r) ; 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) ** inv(share(S,d)) ** inv(share(r,v)); @*/
  public void set(int v);

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

Lst. 4. Contracts for `AtomicInteger`

The postcondition ensures that appropriate new handles for new actions are handed out to the invoking thread. The handle is carrying the role of the calling thread which witnesses its role and its view from the state (last observed value) of `AtomicInteger`. Any instance of a synchronisation 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 synchroniser's reachable states, will therefore not be able to interfere with the threads that participate in this synchronisation.

Handles are specified as `group` without a definition. At the initialisation of the `AtomicInteger`, the constructor issues a full handle for all roles that are passed to the synchroniser. 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 synchronisation.

4.3 Specification

Finally, Lst. 4 shows the complete specification of class `AtomicInteger`. We briefly discuss the method specifications.

The constructor requires the fraction associated to the initial value of the atomic integer. These are the resources that are initially stored inside the synchroniser (`S`), and that can be won by the winning thread in a competition.

Notice that in a cooperative synchronisation mechanism, the resources initially are supposed to be with one of the threads, and the synchroniser is only used as a medium 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 synchronisation, which can be split and passed to all threads that want to access the shared resource.

The contracts of the methods in `AtomicInteger` are all specified based on the specifications we derived in Figure 2 of Section 3. Given the role of the invoking thread, its last visited value from the state (view) and the fraction of `handle`, they all require handles carrying this information. New handles are returned as part of the postconditions. State changing methods, *i.e.* `set` and `compareAndSet`, require that the intended transition is valid, as specified by the `trans` predicate. Finally, the fraction of the resource invariant to be exchanged is specified using `inv` and `share` based on the specifications derived for the basic atomic operations.

4.4 Verification

In verifying client programs using `AtomicInteger`, it is vital to check the definition of `share`, as it should not allow the synchroniser to invent permissions. The distribution defined by `share` should satisfy the following property: *in all states, the total sum of the permissions held by the threads for a resource must not exceed the full permission.* To ensure that the definition of `share` fulfils the condition, we generate proof obligations stating that in any snapshot of the execution, the sum of the fractions assigned to all the threads and the synchroniser must not exceed 1. To show that this proof obligation is respected, we use the definitions of `trans` to extract the set of the valid states, and `share` to determine the resource distribution. The former draws the *maximal state machine* for each role, which shows *all possible transitions* that a role can take. The latter assigns the fraction that each role must hold in each state. Finally, the product of the maximal state machines is exploited to reason about the sum of the shares for each *feasible snapshot*.

Due to space limit, the complete correctness proof of the case studies, including the sanity check of the `share` functions and the proof outline of the programs, are provided in the technical report [1]. Here we only present the correctness of the `findOrPut` method from our `SingleCell` example to illustrate how the specification of `AtomicInteger` works. In Lst. 5 the proof outline of this method demonstrates how the contracts of `AtomicInteger` exchange resources. To show available resources in each step, the outline is annotated with the intermediate states. To instantiate the `SingleCell` class we use these definitions:

```
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 v) { return (r==S && v==E) ? 1 :
           ((r==S && v==D) ? +0: ((r==T && v==D) ? +0:0)); }
```

All the case studies discussed above are verified with our VerCors tool set available at [25]. This tool set is currently being developed to reason about

```
//@ given frac f;
//@ requires handle(T,E,f);
//@ ensures \result == PUT ==> handle(T,D,f) ** PointsTo(data,+0,v);
//@ ensures \result == SEEN ==> handle(T,D,f) ** PointsTo(data,+0,v);
int findOrPut(int v){
{handle(T,E,f) ** 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)) }
    data = v;  //unfold inv(share(S,E)) for Perm(data,1)
{handle(T,W,f) ** PointsTo(data,1,v)}
{handle(T,W,f) ** inv(share(S,D)) ** inv(share(T,W))}
    sync.set(D);
{handle(T,D,f) ** inv(share(S,W)) ** inv(share(T,D)) ** (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))}  // unfold inv(share(T,D))
      if(data == v)
        {handle(T,D,f) ** PointsTo(data,+0,v)}
        return SEEN;
      else
        {handle(T,D,f) ** PointsTo(data,+0,val) ** (val!=v)}
        return COLN;
  }
}
```

Lst. 5. Verification of the `findOrPut` method from `SingleCell`

multithreaded Java programs annotated with permission-based SL. The tool leverages existing verification solutions to multi-threaded Java programs, by encoding verification problems into the Chalice language [13]. The Chalice verifier is then used to prove the translated program correct w.r.t. its specification. All case studies are verified automatically, after providing a few proof hints in terms of intermediate state annotations that we left out here for clarity of presentation. The complete correctness proof of the case studies are presented in the technical report [1] using VerCors syntax which are also available online at [24]. In the presented proof outlines, for clarity, we only annotated the intermediate states of the proof with the predicates that transform resources between the synchroniser and the participating threads.

5 Related Work

Different program logics based on Separation Logic for concurrent programs can be found in the literature. Vafeiadis and Parkinson combined Rely-Guarantee reasoning and SL in RGSep to reason about fine-grained concurrent programs [22]. Assertions in RGSep distinguish between local and shared state, and actions are used to describe the interferences on the shared state between parallel processes. Later, Young *et al.* embedded permission-annotated actions in their assertion language and extended abstract predicates [17] to Concurrent Abstract Predicates

(CAP) [6]. Abstract predicates in CAP encapsulate both resources and interferences, which allows one to reason about the client program without having to deal with all the underlying interferences and resources. The rule for atomics in CAP uses a so called *repartitioning operator*, to extract the resources that the atomic operation requires or ensures.

In CAP it is not possible to reason about synchroniser objects that protect *external* shared resources. Inspired by Jacobs and Piessens [11], and Dodds *et al.* [7], CAP was extended by Svendsen and Birkedal resulting in Higher-Order CAP (HOCAP) [21] and later Impredicative CAP (iCAP) [20] to specify client usage protocols, suitable for synchronisers. iCAP is an important step towards reasoning about synchronisation mechanisms that protect client defined external states.

Ley-Wild and Nanevski [14] proposed Subjective CSL where the thread's *self* view and an *other* view (as a collective effect of the environment) are used to reason about coarse-grained concurrency. Finally, Hobor *et al.* [10] proposed a rule in CSL to reason about programs using barriers as their main synchronisation construct. But they didn't verify the implementation of the barrier.

All techniques mentioned above develop new program logics to reason about concurrent programs. Instead, here, we treat synchronisers at the specification level and we reuse existing verification technology to derive our practical and easy to use specifications from O'Hearn's classical CSL.

6 Conclusion

This paper proposes an approach to specify and reason about atomics as synchronisation constructs. Our approach separates the verification of (1) the correctness of the communication protocol, and (2) the code obeying the protocol, which carries out a rely-guarantee style proof in SL.

Moreover, the paper discusses different patterns to synchronise a set of threads to access a shared resource using atomic read, write and compare-and-set. Based on these patterns, we provide a simple, thread-modular and practical specification of the class `AtomicInteger` from the `java.util.concurrent.atomic` API, using permission-based SL. The specification is easy and intuitive to be used, it only has to be instantiated by: the threads' roles; the shared resources that are protected by the synchroniser; a relation defining allowed state changes; a function that describes for each state change which share of the shared resource is transferred from the thread to the synchroniser, or vice versa; and the handle, as a witness for the provided information.

Using CSL, as a well-established logic, we derived the specification from the standard proof rule for atomic statements. 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 briefly discussed how this can be verified.

We are in the process of extending our approach to shared reading synchronisers, which allows us to verify reference implementations of shared usage synchronisation classes such as `Semaphore`, `ReadWriteReentrantLock` and

`CountDownLatch`, see [2] for preliminary results. As future work, we will also develop a specification of the `AtomicReference`. This will allow us to verify lock-free pointer-based data structures from `java.util.concurrent`.

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

References

1. Amighi, A., Blom, S.C.C., Huisman, M.: Resource protection using atomics: patterns and verifications. Technical Report TR-CTIT-13-10, Centre for Telematics and Information Technology, University of Twente, Enschede (May 2013)
2. Amighi, A., Blom, S., Huisman, M., Mostowski, W., Zaharieva-Stojanovski, M.: Formal specifications for Java's synchronisation classes. In: Lafuente, A.L., Tuosto, E. (eds.) 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 725–733. IEEE Computer Society (2014)
3. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: Palsberg, J., Abadi, M. (eds.) POPL, pp. 259–270. ACM (2005)
4. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003)
5. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G., Leino, K., Poll, E.: An overview of JML tools and applications. STTT 7(3), 212–232 (2005)
6. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 504–528. Springer, Heidelberg (2010)
7. Dodds, M., Jagannathan, S., Parkinson, M.J.: Modular reasoning for deterministic parallelism. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, pp. 259–270. ACM, New York (2011)
8. Haack, C., Huisman, M., Hurlin, C.: Reasoning about Java's reentrant locks. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 171–187. Springer, Heidelberg (2008)
9. Haack, C., Huisman, M., Hurlin, C., Amighi, A.: Permission-based separation logic for multithreaded Java programs (submitted, 2014)
10. Hobor, A., Gherghina, C.: Barriers in concurrent separation logic. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 276–296. Springer, Heidelberg (2011)
11. Jacobs, B., Piessens, F.: Expressive modular fine-grained concurrency specification. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT, POPL 2011, pp. 271–282. ACM, New York (2011)
12. Laarman, A., van de Pol, J., Weber, M.: Boosting multi-core reachability performance with shared hash tables. In: Bloem, R., Sharygina, N. (eds.) FMCAD, pp. 247–255. IEEE (2010)
13. Leino, K., Müller, P., Smans, J.: Verification of concurrent programs with Chalice. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) FOSAD 2007/2008/2009. LNCS, vol. 5705, pp. 195–222. Springer, Heidelberg (2009)
14. Ley-Wild, R., Nanevski, A.: Subjective auxiliary state for coarse-grained concurrency. In: Giacobazzi, R., Cousot, R. (eds.) POPL, pp. 561–574. ACM (2013)
15. O'Hearn, P.W.: Resources, concurrency and local reasoning. Theoretical Computer Science 375(1-3), 271–307 (2007)

16. Parkinson, M.J.: Local reasoning for Java. Tech. Rep. UCAM-CL-TR-654, University of Cambridge, Computer Laboratory (November 2005)
17. Parkinson, M., Bierman, G.: Separation logic, abstraction and inheritance. In: Principles of Programming Languages (POPL 2008), pp. 75–86. ACM Press (2008)
18. Raynal, M.: Concurrent Programming - Algorithms, Principles, and Foundations. Springer (2013)
19. Reynolds, J.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science, pp. 55–74. IEEE Computer Society (2002)
20. Svendsen, K., Birkedal, L.: Impredicative concurrent abstract predicates. In: Shao, Z. (ed.) ESOP 2014. LNCS, vol. 8410, pp. 149–168. Springer, Heidelberg (2014)
21. Svendsen, K., Birkedal, L., Parkinson, M.: Modular reasoning about separation of concurrent data structures. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 169–188. Springer, Heidelberg (2013)
22. Vafeiadis, V., Parkinson, M.: A marriage of rely/guarantee and separation logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007)
23. Vafeiadis, V.: Concurrent separation logic and operational semantics. Electr. Notes Theor. Comput. Sci. 276, 335–351 (2011)
24. Synchronisers in vercors, `https://fmt.ewi.utwente.nl/redmine/projects/vercors-verifier/wiki/Synchronizers`
25. Vercors tool set, `http://www.utwente.nl/vercors/`