Formal Specifications for Java's Synchronisation Classes

Afshin Amighi and Stefan Blom and Marieke Huisman
and Wojciech Mostowski and Marina Zaharieva-Stojanovski
Formal Methods and Tools, University of Twente, The Netherlands
Email: {a.amighi,s.blom,m.huisman,w.mostowski,m.zaharieva}@utwente.nl

***Abstract*—This paper discusses formal specification and verification of the synchronisation classes of the Java API. In many verification systems for concurrent programs, synchronisation is treated as a primitive operation. As a result, verification rules for synchronisation are hard-coded in the logic, and not verified. These rules describe the concrete semantics of the given synchronisation primitive, and manage how resources are protected by synchronisation.**

In contrast, this paper describes several synchronisation primitives at the *specification* level, by specifying the behaviour of synchronisation routines from the Java API at method level using permission-based Separation Logic. This gives a generalised, high-level, and easily extendable approach to formalisation of arbitrary synchronisation mechanisms, which allows for modular treatment of synchronisation in verification. Notably, our approach does not only apply to locks, but also to other synchronisation mechanisms such as semaphores and latches that we also discuss in the paper. Finally, we used the verification tool that we are developing and successfully verified (so far simplified) reference implementations of all presented synchronisers; the paper discusses the verification of one of them.

I. Introduction

Concurrency is globally present in practically all modern production software. The complexity of concurrent program behaviours makes it challenging to guarantee correct functioning of such software. This has led to a strong demand for formal verification systems to deal with concurrent programs. As a result, several techniques and verification tools [1], [2], [3], [4] have emerged that provide automated reasoning for realistic concurrent software. Many of the underlying program logics employed in these recent verification systems share the following characteristics: (i) they are based on the notion of read and write permissions to access heap locations [5], [6], and (ii) they explicitly handle the program heap [7] (typically using or inspired by Separation Logic [8]). These ingredients are sufficient to support thread-modular verification, i.e., verification of a single thread in isolation. Moreover, successful verification of all program threads annotated with permissions guarantees that the concurrent program is free of data races.

The VerCors project concentrates on the application of permission-based Separation Logic to multi-threaded Java programs [9], [10]. It focuses in particular on the concurrent data structures of the Java API, where the goal is to specify and verify full functional correctness properties, rather than to reason only about interference between multiple threads.

Locking and other synchronisation mechanisms are an essential part of these concurrent data structures (and other concurrent programs). Many verification systems for concurrent programs consider locking as a *primitive* operation of the language, and the reasoning system provides explicit rules for handling this. Also in our initial work, we formalised Java's reentrant locks as primitive operations in the logic [11].

The core of almost any lock formalisation is the notion of a *resource invariant* [1] that makes the implicit information about the resources (heap locations) protected by the lock explicit by defining the access rights to the locations that are protected by the lock. Based on this notion, the verification of a program using locks essentially boils down to checking permission transfer: upon locking, permissions are transferred from the lock's resource invariant to the locking thread; upon unlocking, the thread transfers all the permissions back into the lock. Only the thread that currently holds the permissions can access the protected location.

This paper shows how the treatment of resource invariants is lifted to the API level of Java, i.e., we provide a *specification-based* approach to reason about locks. To make the approach applicable to different synchronisation mechanisms, we generalise the notion of a lock, i.e., we consider any routine that uses synchronisation to transfer a set of permissions as a *locking routine*. With our approach, we can specify arbitrary synchronisation mechanisms from the Java API in a similar way, and provide the ability to reason with these specifications modularly. The overall added value of our approach is the consequent ability to reason about arbitrary concurrent Java programs. This supports the goals of the VerCors project: Java concurrent data structures use the API locks extensively, hence we need to have them specified and verified in order to verify the correctness of any of the data structure implementations.

Concretely, this paper illustrates our approach by presenting Separation Logic specifications for the following synchronisation classes: the reentrant read-write family of locks, semaphore, and the count-down latch. This selection is guided by the results of an analysis of the Qualitas Corpus benchmark suite [12] using the Histogram tool [13], identifying the most-often used classes of the java.util.concurrent API.

An on-going part of our project is the development of an automated tool set for our logic. For the work presented here we already use our tool to a large degree; slightly simplified reference implementations of all the discussed synchronisers

have been successfully verified w.r.t. our specifications. These verified implementations are all available on-line together with a web-based version of our tool [14].

The rest of the paper is organised as follows. Sect. II provides background on permission-based Separation Logic, and a short description of the synchronisation classes from the Java API. Sect. III discusses the specification of the synchronisation classes mentioned above. Sect. IV discusses the verification of one of our reference implementations. Finally, Sect. V concludes the paper with plans for the future and related work.

II. Background

This section briefly introduces permission-based Separation Logic, and how it supports reasoning about multi-threaded Java programs, and in particular reentrant locks. For more information, we refer to [10], [11], [15]. It also provides some background information on the Java concurrency API.

A. Resource Protection with Permissions

Separation Logic originally has been proposed as an extension of Hoare Logic to reason about mutable data structures [8]. However, it is also suitable to reason about multi-threaded programs [1], because it allows one to reason explicitly about the heap, and to identify which part of the heap is affected by a thread. This paper uses *permission-based Separation Logic*, a variant of Separation Logic where access to a location is decorated with a *read* or *write permission*.

In classical Hoare Logic, assertions are properties over the state. In Separation Logic, the state is explicitly divided into the heap, where all object information is stored, and the store, containing information related to the current method call. We distinguish between *resource expressions* (R, typical elements r_i) and *logical 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:

$$
\begin{array}{lcl}
R & ::= & b \mid \mathsf{PointsTo}(\mathsf{field}, \mathsf{frac}, e) \\
 & \mid & \mathsf{Perm}(\mathsf{field}, \mathsf{frac}) \mid (\textbf{\textbackslash forall} * \ \mathsf{T}\ \mathsf{v}; b; r) \\
 & \mid & r_1 \ast\ast r_2 \mid b_1 ==> r_2 \mid e.\mathsf{P}(e_1, \cdots, e_2) \\
E & ::= & \text{any } \textit{pure} \text{ expression} \\
B & ::= & \text{any } \textit{pure} \text{ expression of type boolean}
\end{array}
$$

where T is an arbitrary type, v is a variable name, P is an abstract predicate [16] of a special type **resource**, field is a field reference, and frac denotes a fractional permission.

Intuitively, an assertion $\mathsf{PointsTo}\,(e.f, \pi, v)$ (or $e.f \overset{\pi}{\mapsto} v$ in classical notation) 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. When the value is not important, we often use Perm, using that $\mathsf{PointsTo}(e.f, \pi, v)$ is equivalent to $\mathsf{Perm}(e.f, \pi)$ && $e.f == v$. A formula $\phi_1 \ast\ast \phi_2$ holds for a heap if the heap can be split into two *disjoint* heaps, with the first sub-heap satisfying ϕ_1, and the second sub-heap satisfying ϕ_2. Finally, assertions can use abstract predicates P to encapsulate the state space [16]. Below, we sometimes use an additional requirement that the abstract predicate is a **group** [10], *i.e.*, it can be split over permissions.

Permissions π are values in the domain $(0, 1]$. Each thread always holds a set of permissions on locations. If a thread has a full permission i.e., the value 1, for a location, then it has permission to *write* this location. If a thread has a fractional permission, i.e., a fraction less than 1, then it has a *read* permission for this location. 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 a location, and whenever a thread has a read permission, all other threads holding a permission on this location simultaneously can have a read permission only. This ensures that verified programs are *data-race-free*. Permissions can be split and combined, to change between read and write permissions, and they can be transferred between threads upon thread *creation*, upon *joining* a terminated thread, and by *synchronisation*. The modularity in reasoning is supported by the semantics of the separating conjunction. Exclusive access to a location guarantees that the location is not aliased with other locations on the heap and thus can be reasoned about in isolation. Partial access, on the other hand, guarantees the read-only property for the location, i.e., its value never changes in the scope of a non-exclusive access permission. In turn, this ensures that we do not have to reason about thread interference.

The concrete syntax of our specifications is a combination of permission-based Separation Logic with the Java Modeling Language (JML)[1] [17], including features like ghost fields. 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. Building on the JML annotation language allows us to specify permission access properties side by side with complex functional properties. In the scope of the synchronisation classes, however, the permissions are the main focus.

B. Reasoning about Built-in Locks

As mentioned above, permissions are transferred upon synchronisation. We briefly describe the logic to reason about *built-in* Java reentrant locks developed by Haack et al. [11], which forms the basis of the work in this paper. We lift this logic to specification-level and generalise it to other synchronisation mechanisms in the Java API.

Following O'Hearn [1], for each lock, an abstract predicate inv describing the *resource invariant* is specified, describing which locations are protected by the lock. Whenever a lock is acquired for the first time, the locking thread obtains these resources, and thus can access the data protected by the thread. Upon final release of the lock, the thread is forced to give up the resources. To distinguish initial acquirings and final

[1]Hence we also diverted from the classical Separation Logic notation of $*$ for the separating conjunction to $**$ in order to maintain the multiplication operator.

$$\frac{\Gamma \vdash u, S : \mathsf{Object}, \mathsf{lockset}}{\Gamma; v \vdash \begin{array}{c}\{\mathsf{LockSet}\,(S) \ast u \notin S \ast u.\mathsf{init}\} \\ u.\mathsf{lock}\,() \\ \{\mathsf{LockSet}\,(u \cdot S) \ast u.\mathsf{inv}\}\end{array}}\ (\text{Lock})$$

$$\frac{\Gamma \vdash u, S : \mathsf{Object}, \mathsf{lockset}}{\Gamma; v \vdash \{\mathsf{LockSet}\,(u \cdot S)\}u.\mathsf{lock}\,()\{\mathsf{LockSet}\,(u \cdot u \cdot S)\}}\ (\text{Re-Lock})$$

Fig. 1. Proof rules for initial and reentrant acquiring of a lock from [11].

releases from reentrant acquirings and releases, each thread maintains a multi-set LockSet that keeps track of all locks (including their multiplicity) that the thread currently holds. The lock sets are necessary to properly treat *lock reentrancy*: if a thread acquires a lock that is already in the lock set, it does not obtain any permissions, and if a thread releases a lock, it does not have to give up any permissions if the lock afterwards is still in the lock set.

In addition, Haack et al. developed some technical machinery for lock *initialisation*. A lock can only be used when it has been initialised, *i.e.*, the access permissions specified in the resource invariant are stored "into" the lock. Lock initialisation is indicated by a specification-only commit statement: the proof rule for commit ensures that the permissions from the thread are *absorbed* into the lock, and the lock is ready to be used.

To give a flavour of this logic, Fig. 1 presents the proof rules from [11] for initial and reentrant acquiring of a lock. The first rule states that if a thread locks u, and the set of currently held locks does not contain u yet, then upon completion of the u.lock() statement, u is in the lock set, and the resource invariant has been transferred to the thread holding the lock on u. However, as indicated by the second rule, if u is already held by the current thread, no permissions are transferred, only bookkeeping of the additional acquiring of the lock is done. Note that here the statement u.lock() is a built-in instruction of the language. Reasoning about the Java **synchronized** statement requires a straightforward translation to suitable o.lock() and o.unlock() statements. This paper shows that this approach generalises to the other synchronisation classes from the Java API as well.

However, in our approach, resource invariants may be parametrised with a fraction expression, to support both write and read-only locking scenarios with just one predicate, i.e., instantiating the predicate with a concrete value makes the resource invariant describe a write permission, a read permission, or a mixture of both. Class Object declares a default predicate inv, setting it to true. This definition can be *referred to* and *extended* in subclasses and interfaces.

C. The Java Concurrency API

Basic concurrency support in Java is provided by the Thread class. Further, every object can function as a lock, using the **synchronized** keyword, which makes it impossible to forget to release a lock. However, this also puts strong restrictions on the design and implementation of (bigger) systems; in particular, its syntactic limitations make it impossible to acquire and release locks at arbitrary points in the code.

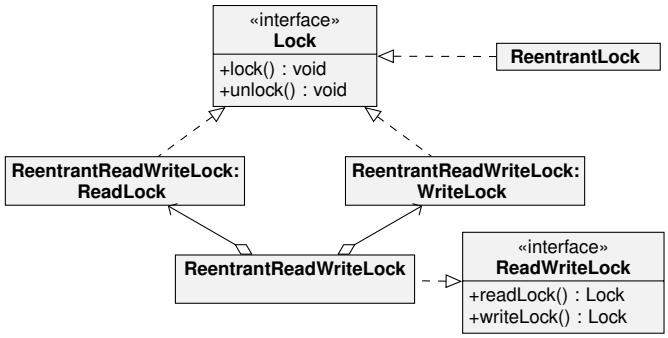

Fig. 2. The hierarchy of locks in the java.util.concurrent package.

This has been addressed by the Java concurrency package java.util.concurrent introduced in Java 1.5 [18]. Among other things, this package features: (i) locks and other synchronisation primitives, (ii) the Executor framework, providing task-based parallelism, (iii) thread-safe data structures, such as maps and queues, and (iv) support for atomic variables.

In its basic form, the lock, specified by the Lock interface, provides either exclusive or controlled shared access to a shared resource. For situations where, depending on the execution context, either shared or exclusive access is required, the API defines a ReadWriteLock interface. All interfaces are implemented by classes that support lock reentrancy. Fig. 2 shows the complete hierarchy of Java's lock classes.

In addition, the concurrency package contains implementations of several other synchronisation classes, such as a semaphore, a count-down latch, and a cyclic barrier.

To find out which classes are most often used, and where thus our specifications efforts are most useful, we counted the number of references to classes in the concurrency packages for each of the projects in the standard Qualitas Corpus benchmark suite [12] using our Histogram tool [13]. Fig. 3 shows which synchronisation classes are in the top 25 of most-used classes. The analysis further shows that the reentrant family of locks is used most often: 22 out of 29 systems that use locks reference the ReentrantLock class.

III. Specifications of the Synchronisation Classes

This section presents Separation Logic specifications for several synchronisation classes from the Java concurrency API. As mentioned above, the starting point for our specifications is the earlier formalisation of reentrant Java locks [11], [15], but we lift all elements of that formalisation to the specification layer. Therefore, the underlying concepts can be shared,

Place	#Refs	#Projects	Class name
5	644	22	locks.ReentrantLock
10	1170	18	locks.ReentrantReadWriteLock
11	2633	17	locks.Lock
14	387	15	CountDownLatch
18	534	13	locks.ReadWriteLock

Fig. 3. The synchronisation classes in the top 25 of most used concurrency classes in the java.util.concurrent package.

reused, and adapted by different synchronisation classes of the Java concurrency API. In particular, the notion of a resource invariant is at the base of all the specifications. In essence, all specifications of the synchronisation methods express how permissions are transferred between the thread and the resource invariant. This describes the essential differences between the synchronisation mechanisms.

Below, we first discuss how lock initialisation is treated at specification level. Then we show how the logic for built-in reentrant locks from [11] is generalised to specify the Lock interface, and how the specifications of ReentrantLock and ReadWriteLock both are built on top of this general specification for Locks. Moreover, to illustrate that also other synchronisation mechanisms can be specified using the same approach, we discuss specifications of two other frequently used synchronisation classes: Semaphore and CountDownLatch.

A. Initialisation of Resource Invariants

As mentioned above, resource invariants have to be initialised, *i.e.*, the permissions have to be transferred "into" the synchroniser, before the synchronisation mechanism can be used. This ensures that the resources can be passed to a user upon synchronisation without introducing new resources.

Initialisation of the resource invariant is done in the same way for all synchronisation mechanisms: class Object declares a boolean field initialized that tracks information about the initialisation state of the resource invariant. Newly created locks are not initialised; the specification-only method commit can be used by the client code to irreversibly initialise the lock. This means that the resources protected by the lock, as specified in inv, become shared. To achieve this, commit requires the client to provide the complete resource invariant inv(1), together with an exclusive permission to change initialized. The method consumes the invariant ("stores it into the lock"). Moreover, it ensures that initialized cannot be changed anymore by consuming part of the permission to access this field, effectively making it read-only. For convenience, the result of commit is encapsulated in a single resource predicate initialized, which can be passed around and used as a permission ticket for locking operations, see below.

```
//@ ghost boolean initialized = false;
//@ group resource initialized(frac p) = PointsTo(initialized, p/2, true);

//@ requires inv(1) ∗∗ PointsTo(initialized, 1, false);
//@ ensures initialized(1);
public void commit();
```

The default location for the call to commit is at the end of the constructor of the synchronisation object. More complex lock implementations (that we do not discuss in this paper) may require moving this call to another location in the program.

The actual resource invariant is typically decided by the user of the synchronisation class, therefore it is passed as a class parameter. For example, given a two-point coordinate class using a ReentrantLock, the resource invariant that protects the x coordinate (only) is specified with xInv, which is passed both during type declaration and during instantiation of the lock:

```
//@ resource xInv(frac p) = Perm(x, p);
Lock/∗@< xInv, ... >@∗/ xLock = new ReentrantLock/∗@<xInv>@∗/();
```

As mentioned in Sect. II, in our specifications such parameters (of which there will be more, hence the "…" above) are received through variables specified with the **given** keyword.

B. Lock Hierarchy Specification

The synchronisation classes in the Lock hierarchy in the concurrency package (see again Fig. 2) are devoted to resource locking scenarios where either full (write) access is given to one particular thread or partial (read) access is given to an indefinite number of threads. We first discuss the specification of the Lock interface, and then we proceed with specifications of different lock implementations.

1) Lock Interface Specification: As explained above, our specification approach of the synchronisation mechanisms is inspired by the logic of Haack et al. [11]. However, we cannot just translate the rules from this logic (as in Fig. 1) into method specifications of the Lock interface, because the Lock interface can be used in different and wider settings than considered by Haack et al. In particular, Lock implementations may be non-reentrant; they may be used to synchronise non-exclusive access; and they may be used in *coupled* pairs to change between shared and exclusive mode (see the read-write lock specification below). Therefore, compared to the work of Haack et al., the following changes for the specification given in Lst. 1 are necessary:

- The locks are parametrised by boolean variables isExclusive and isReentrant, which can be correspondingly instantiated by implementations.
- To allow non-exclusive synchronisation, resource invariants have to be groups, see Sect. II-A.
- For the non-exclusive locking scenarios the client program has to record the amount of the resource fraction that was obtained during locking, so that the lock can reclaim the complete resource fraction upon unlocking. This information is passed around in the held predicate, which holds this fraction. This is purposely not declared as a group, so that clients are obliged to return their whole share of resources. The held predicate is returned during locking in exchange for the initialized predicate which is temporarily revoked for the time that the lock is acquired.
- For situations where several locks share the same resource and are effectively coupled as one lock, we need to ensure that only one lock is locked at a time. The coupling itself is realised by holding a reference to the parent object that maintains the coupled locks. The exclusive use of coupled locks is ensured by storing and checking this parent object in the set of currently held locks.
- A separate specification case is provided for reentrant locking, if the isReentrant flag is set.

As a result, in the specification of method lock() (Lst. 1, line 9–17), when the lock is acquired for the first time, the locking thread gets permissions from the lock. If the lock is reentrant, and the thread already holds the lock, then no new permission is gained, only the multi-set of locks held by the current thread

```
//@ given group (frac −> resource) inv;
//@ given boolean isExclusive, isReentrant;
public interface Lock {
 //@ group resource initialized(frac p);
 //@ resource held(frac p);

 //@ ghost public final Object parent;

 /*@ given bag<Object> S, frac p;
   requires LockSet(S) ** !(S contains this) ** initialized(p);
   requires parent != null ==> !(S contains parent);
   ensures LockSet(this::parent::S) **
    inv(isExclusive ? 1 : p) ** held(p);
  also
   requires isReentrant ** LockSet(S) **
    (S contains this) ** held(p);
   ensures LockSet(this::S) ** held(p); @*/
 void lock();

 /*@ given bag<Object> S, frac p;
   requires LockSet(this::S) ** (S contains this) ** held(p);
   ensures LockSet(S) ** held(p);
  also
   requires held(p) ** inv(isExclusive ? 1 : p);
   requires LockSet(this::parent::S) ** !(S contains this);
   ensures LockSet(S) ** initialized(p); @*/
 void unlock();
}
```

Lst. 1. Specification of the Lock interface.

is extended with this lock (where :: denotes bag addition). For coupled locks (where the parent is not null) the presence of the parent in the lock set is also checked and recorded, to prevent parallel use of the coupled locks. The specification of method unlock() describes the reverse process (Lst. 1, line 20–26): if the multi-set of locks contains the specific lock only once, then this means the return of permissions to the lock (*i.e.*, inv does not hold in the postcondition) according to the held predicate; otherwise, the thread keeps the permissions, but one occurrence of the lock is removed from the multi-set.

2) ReentrantLock Specification: Class ReentrantLock implements the Lock interface as an exclusive, reentrant lock. Thus, it inherits all specifications from Lock and appropriately instantiates the class parameters isReentrant and isExclusive:

```
//@ given group (frac −> resource) inv;
class ReentrantLock implements Lock /*@< inv, true, true >@*/ {
```

3) ReadWriteLock Specification: The ReadWriteLock is not a lock itself, but a wrapper of two coupled Lock objects: one of them provides exclusive access for writing (WriteLock), while the other allows concurrent reading by several threads (ReadLock). The two classes are commonly implemented as inner classes of the class that implements the ReadWriteLock interface (see Fig. 2 on page 3). The two locks are intended to protect the same memory resources. Hence our specifications in Lst. 2 state that the two getter methods for obtaining the two locks return a lock object with the same resource inv, but which are exclusive and non-exclusive, respectively. The aggregate read-write lock has to be initialised itself. Further, we state in the respective postconditions of the getter methods that the obtained locks are initialised and hence can be acquired, and that they have the same parent object, which is an instance of

```
//@ given group (frac −> resource) inv;
//@ given boolean reentrant;
interface ReadWriteLock {
 //@ group resource initialized(frac p);

 /*@ given frac p;
   requires initialized(p);
   ensures \result.parent == this ** \result.initialized(p); @*/
 /*@ pure @*/ Lock /*@< inv, false, reentrant >@*/ readLock();

 /*@ given frac p;
   requires initialized(p);
   ensures \result.parent == this ** \result.initialized(p); @*/
 /*@ pure @*/ Lock /*@< inv, true, reentrant >@*/ writeLock();
}
```

Lst. 2. Specification of the ReadWriteLock interface.

the class implementing the ReadWriteLock interface.

C. Semaphore Specification

The Semaphore class represents a *counting semaphore*. It is used to control threads' accesses to a shared resource, by restricting the number of threads that can access a resource simultaneously. Each semaphore is provided with a property *permits*, that represents the maximum number of threads that can access the protected resource. Accessing the resource must be preceded by acquiring a permit from the semaphore. A semaphore with n permits allows a maximum of n threads to access the same resource simultaneously. If n threads are holding a permit, a new thread that tries to acquire a permit blocks until it is notified that a permit is released.

When initialised with more than 1 permit, a semaphore closely corresponds to a non-reentrant ReadLock, but with the number of threads accessing the shared resource explicitly stated and controlled. When initialised with 1 permit, it provides exclusive access, and behaves the same as a non-reentrant WriteLock. Therefore, the specification of the semaphore is a stripped-down version of the Lock specification (see Lst. 3). In particular, semaphores are never reentrant, and they are not used in coupled combinations. Moreover, since the maximum number of threads that can access the shared resource is predefined with the permits field, we can also limit ourselves to simply providing each acquiring thread with an equal split of 1/permits of the resource invariant. Note also that there is no access permission required for the permits field as it is declared to be final and hence can never change after initialisation.

D. CountDownLatch Specification

Essentially, a count-down latch is a *distributed* multi-thread lock. Typically, a parent thread initialises a latch with a count and then passes it to a number of worker threads together with some shared resource for the threads to work on. Each worker thread, once finished, calls method countDown() on the latch to signal that it releases its share on the resource. Threads can wait for all worker threads to finish by calling the blocking await() method. Each call to countDown() decreases the internal latch counter, and once this reaches zero, all awaiting threads unblock and can use the protected resource again.

```
//@ given group (frac −> resource) inv;
public class Semaphore {
  //@ resource held(frac p);

  //@ ghost final int permits;

  //@ requires inv(1) ∗∗ permits > 0;
  //@ ensures  initialized(1) ∗∗ this.permits == permits;
  public Semaphore(int permits);

  //@ given frac p;
  //@ requires initialized(p);
  //@ ensures inv(1/permits) ∗∗ held(p);
  public void acquire();

  //@ given frac p;
  //@ requires inv(1/permits) ∗∗ held(p);
  //@ ensures initialized(p);
  public void release();
}
```

Lst. 3. Specification of the Semaphore class.

```
//@ given group (frac −> resource) inv;
public class CountDownLatch {
  //@ resource held(frac p);

  //@ requires count > 0;
  //@ ensures initialized(1);
  //@ ensures (\forall∗ int i; 0 <= i && i < count; held(1/count));
  public CountDownLatch(int count);

  //@ given frac p;
  //@ requires held(p) ∗∗ inv(p);
  public void countDown();

  //@ given frac p;
  //@ requires initialized(p);
  //@ ensures  inv(p);
  public void await();
}
```

Lst. 4. Specification of the CountDownLatch class.

In the context of our work, the await() is technically a *locking* operation – permissions are transferred to the calling thread – while countDown() is an *unlocking* operation – the calling thread gives up permissions for resources it worked on. Thus, intuitively, compared to the lock, the flow of resource permissions is reversed. Moreover, latches are distributed: each worker thread performs only a partial unlock to collectively achieve a full one, and all awaiting threads do a distributed lock. A high-level specification for the latch that can account for all possible resource redistribution scenarios between n worker threads that call countDown and a different number m of awaiting threads is difficult to achieve without introducing major complexities. Therefore, here we only discuss one basic scenario and only sketch ideas for a more general solution.

Our basic scenario assumes that each worker thread gets an equal non-exclusive share of some resource, and awaiting threads receive a fractional split of the whole resource. Note that if there is only one awaiting thread, it can reclaim the whole resource invariant protected by the latch. The associated specifications are shown in Lst. 4. When the latch is constructed (lines 5–8) two predicates are returned: initialized(1), reflecting the ability of the receiving thread to call await, plus count number of equal held predicates. The thread that created the latch should pass these to each of the worker threads, along with a corresponding split of the resource invariant, i.e., it is not the responsibility of the latch to distribute the resource invariant at this point. When calling countDown (lines 10–12), the worker thread presents its held predicate and an associated fraction of the resource invariant, which is then consumed back into the latch. The await method (lines 14–17) expects at least a fraction of the initialized predicate and returns an associated split of the resource invariant. In case there is only one awaiting thread and the initialized predicate is unsplit, the complete resource invariant is returned.

A generalised usage scenario for the latch requires to arbitrarily split the resource invariant for the worker threads, e.g., in such a way that each worker has an exclusive access to part of the resource, rather than a shared access to the whole resource. Upon completion of all n countDown calls, the resource invariant is reconstructed into a full one and then split again according to another division schema for the m awaiting threads. The core idea to solve this is to provide separate splits of the resource invariant for the countDown and await operations, and carry around the identity of the corresponding calling thread in an integer parameter, resulting in sets of cdPred(i) and awPred(j) predicates such that:

```
inv(1) ∗−∗ (\forall∗ int i; 0 <= i && i < nCountDown; cdPred(i))
inv(1) ∗−∗ (\forall∗ int j; 0 <= j && j < nAwait; awPred(j))
```

where $∗−∗$ denotes a separating equivalence (two-way magic wand [8]), while nCountDown and nAwait correspond to the numbers of worker and awaiting threads, respectively.

IV. Verification of Synchronisation Classes

To show adequacy of our specifications, they should be verified w.r.t. the Java reference implementation. All synchronisation classes of the java.util.concurrent package, including the ones discussed here, are implemented on top of a generic synchronisation framework: Doug Lea's family of AbstractSynchronizer classes [19]. This framework provides (1) the actual, generic synchronisation facility, implemented in terms of compare-and-set operations over atomic integers, and (2) fairness control, implemented in terms of thread queues.

We are currently developing an automated tool set, called VerCors, for the verification of Java programs annotated with permission-based Separation Logic. Our tool leverages existing verification solutions to multi-threaded Java programs; in particular, we encode our verification problems into the Chalice language [2] and then use the Chalice verifier to prove the translated programs correct w.r.t. their specifications. Although Chalice is based on implicit dynamic frames, instead of Separation Logic, its support for concurrency, resource invariants and permission transfer, and the fact that it is fully automated, makes it very suitable for our purpose.

As our tool is under development, it is not yet able to deal with the full specifications as presented in this paper – notably, it cannot reason about LockSet predicates directly yet – or with the complexity of the actual Java reference implementations of

```
//@ given group (frac −> resource) inv;
public class SimpleReentrantLock {
 //@ resource state(int id, int lev) = (lev > 0) ==> Perm(count,1);
 //@ boolean trans(int c, int n) = (c==U && n>0) || (c>0 && n==U);
 //@ frac share(int val) = val==U ? 1 : 0;

 private final int U = 0;
 private AtomicInteger/*@<inv, share, trans>@*/ sync;
 private int count = 0;

 //@ requires inv(1);
 //@ ensures (\forall* int t; t>0 ; state(t, 0));
 public SimpleReentrantLock(){
  sync = new AtomicInteger/*@<inv, share, trans>@*/ (U);
 }

 //@ given int lev;
 //@ requires state(tid, lev) ** tid > 0;
 //@ ensures state(tid, lev+1) ** (lev == 0 ==> inv(1));
 public void lock(int tid){
  int cur = sync.get() ;
  if (tid == cur) { count++; } else {
   while (!sync.compareAndSet(U, tid));
   count = 1;
  }
 }

 //@ given int lev;
 //@ requires state(tid, lev) ** tid>0 ** lev>0 ** inv(1);
 //@ ensures state(tid, lev−1) ** (lev > 1 ==> inv(1));
 public void unlock(int tid){
  if (tid == sync.get())
   if (count == 1) {
    count = 0; sync.set(U);
   }else if (count > 1) { count−−; }
 }
}
```

Lst. 5. A simplified reentrant lock implementation and specification.

the synchronisation classes. Therefore, this section discusses verification of a simplified reference implementation: on the specification level, we explicitly encode LockSet predicates, and on the implementation level, we only verify the actual synchronisation behaviour, and we ignore fairness control. However, we believe that the approach as discussed here can be generalised in the future to verify the complete reference implementation w.r.t. the complete specification.

Concretely, here we discuss the verification of the correctness of a reentrant lock implementation as shown in Lst. 5. The fully annotated version, and the verified Semaphore along with a simplified CountDownLatch implementation can be found on-line [14]. This webpage also gives access to a web-based interface for the VerCors tool.

As mentioned, our implementation is inspired by Lea's abstract synchronisers framework, but abstracts away everything that is related to fairness control. This allows us to use AtomicInteger as the synchronisation primitive. Also, instead of using the LockSet predicate, the specification in Lst. 5 encodes lock reentrancy using a predicate state that specifies the reentrancy level (denoted with lev) of the thread identifier tid. This encodes the same behaviour for reentrant, mutually exclusive locks as the specification in Lst. 1: the number of times a reentrant lock occurs in the lock set of a thread tid is

```
//@ given group (frac −> resource) inv;
//@ given (int −> frac) share;
//@ given (int, int −> boolean) trans;
public class AtomicInteger {

 //@ requires inv(share(v));
 public AtomicInteger(int v);

 public int get();

 //@ given int o;
 //@ requires trans(o, v) ** inv(share(v));
 public void set(int v);

 //@ requires trans(x, n) ** inv(share(n) − share(x));
 //@ ensures \result ==> inv(share(x) − share(n));
 //@ ensures !\result ==> inv(share(n) − share(x));
 public boolean compareAndSet(int x, int n);
}
```

Lst. 6. A specification for AtomicInteger as a synchronisation primitive.

resource equivalent to the state(tid, lev) predicate.

Due to space restrictions, we cannot discuss the specification of AtomicInteger as a synchronisation primitive in full detail; we only summarise its most important aspects; for more details we refer to [20]. In the case we consider here, the AtomicInteger controls a competitive synchronisation scheme: all threads compete to obtain access to the resource protected by the lock. Lst. 6 presents the AtomicInteger specification (simplified from the fully general specification for readability). The specification is parametrised by a relation trans describing the possible state transitions in the synchronisation protocol, and a function share specifying which share of the resource invariant (possibly 0) is transferred between the synchroniser and the thread[2]. The contracts of compareAndSet exchange the difference between the permissions assigned for expected value and new value. Subtraction of two fractions is a cut-off subtraction operator: $(p \dot{-} q)$ is 0 if $q > p$, $(p - q)$ otherwise.

The value of the atomic integer sync in Lst. 5 encapsulates the state of the lock: when positive, it denotes the thread that holds the lock; when zero (constant U) the lock is free. When the sync object is constructed, the resource invariant is transferred into this object. By successfully calling lock, a thread obtains the resource invariant, and it transfers this back to the synchroniser by calling unlock. The predicate state expresses that if a thread holds the lock, then it has full permission to update the reentrancy level (line 3) stored in the field count. The trans relation is instantiated to express that if the lock is free, it can be acquired, and if the lock is held, it has to be released before another thread can acquire it. The share function ensures correct permission transfer: in the lock method, if the call to compareAndSet(U, tid) is successful, thread tid obtains a full share of the resource invariant. When releasing the lock, by calling sync.set(U) in unlock(), the thread has to give up its share of the resource invariant.

Finally, it is easy to see that this implementation correctly handles the lock reentrancy level, stored in count. If a thread

[2]The tool checks some additional constraints on these arguments, to ensure the specification does not create new resources.

calls lock() while holding the lock (tid == cur), it simply updates the reentrancy level. Any thread acquiring a lock for the first time stores its thread identifier in sync by successfully calling the atomic compareAndSet operation (line 23), followed by assigning 1 to count. The unlock() operation ensures that the lock only becomes available when count == 1, setting it to U first. Otherwise, count is only decreased by 1.

The verification of the class SimpleReentrantLock w.r.t. its specification is done fully automatically by our tool set, after providing a few additional proof hints in terms of intermediate state annotations.

V. Conclusions

Based on examples of the family of Java locks, the semaphore, and the count-down latch from the Java API, we present a generalised approach for handling synchronisation primitives in permission-based Separation Logic for concurrent Java. We lift all mechanisms associated with synchronisation handling, and the corresponding permission transfer, to the specification layer of the logic. This way we provide a modular verification mechanism that is applicable to arbitrary concurrent Java programs, and we enable the verification of the synchronisation routine implementations themselves. We also discuss the verification of a reference implementation of a reentrant lock, focusing on the exclusive access (with reentrancy) aspects of the implementation only. Finally, we have also verified reference implementations of Semaphore and CountDownLatch [14].

The work presented here extends our earlier formalisation of reentrant locks [11]. Several other built-in formalisations of locks and synchronisation primitives exist. The Chalice system [2] formalises simple non-reentrant locks built into the Chalice language. The work of Gotsman et al. [21] is similar to our earlier formalisation, and we believe that our high-level approach could also be easily applied there to treat a wider range of synchronisation primitives. Similarly, the work of Hobor and Gherghina on formalising barriers in Separation Logic [22] follows very similar principles to ours and we believe that their formalisation could be adapted to fit our specification framework too. Finally, the VeriFast tool [3] adopts an approach similar to ours – simple non-reentrant locking routines are also specified on the API level, and so-called higher-order abstract predicates are functionally similar to our class level specification parameters.

For future work, following what we discussed in Sect. III-D, we will improve our ideas for a more flexible permission splitting and (re-)distribution mechanism. We also plan to adapt other permissions systems to our work, e.g., "fractionless" permissions [23], or tree permissions [24]. Then, we will investigate how to verify an implementation of a coupled read-write lock, where possibly two cooperating instances of an AtomicInteger are to be used for synchronisation. Finally, we will keep on improving our tool to enable verification of more realistic case studies and attempt to verify some of the Java implementations for concurrent data structures.

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

References

[1] P. W. O'Hearn, "Resources, concurrency and local reasoning," *Theoretical Computer Science*, vol. 375, no. 1–3, pp. 271–307, 2007.

[2] K. Leino, P. Müller, and J. Smans, "Verification of concurrent programs with Chalice," in *Lecture notes of FOSAD*, ser. LNCS, vol. 5705. Springer, 2009.

[3] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens, "VeriFast: A powerful, sound, predictable, fast verifier for C and Java," in *NASA Formal Methods*, ser. LNCS, vol. 6617. Springer, 2011, pp. 41–55.

[4] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "VCC: A practical system for verifying concurrent C," in *Theorem Proving in Higher Order Logics (TPHOLs)*, ser. LNCS, vol. 5674. Springer, 2009, pp. 23–42.

[5] J. Boyland, "Checking interference with fractional permissions," in *Static Analysis Symposium*, ser. LNCS, R. Cousot, Ed., vol. 2694. Springer, 2003, pp. 55–72.

[6] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson, "Permission accounting in separation logic," in *Principles of Programming Languages*, J. Palsberg and M. Abadi, Eds. ACM, 2005, pp. 259–270.

[7] M. J. Parkinson and A. J. Summers, "The relationship between separation logic and implicit dynamic frames," *Logical Methods in Computer Science*, vol. 8, no. 3:01, pp. 1–54, 2012.

[8] J. Reynolds, "Separation logic: A logic for shared mutable data structures," in *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*. IEEE Computer Society, 2002, pp. 55–74.

[9] A. Amighi, S. Blom, M. Huisman, and M. Zaharieva-Stojanovski, "The VerCors project: Setting up basecamp," in *Programming Languages meets Program Verification (PLPV 2012)*. ACM, 2012, pp. 71–82.

[10] C. Haack and C. Hurlin, "Separation logic contracts for a Java-like language with fork/join," in *Algebraic Methodology and Software Technology*, ser. LNCS, J. Meseguer and G. Rosu, Eds., vol. 5140. Springer, 2008, pp. 199–215.

[11] C. Haack, M. Huisman, and C. Hurlin, "Reasoning about Java's reentrant locks," in *6th Asian Conference on Programming languages and Systems*, ser. LNCS, G. Ramalingam, Ed., vol. 5356. Springer, 2008, pp. 171–187.

[12] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "Qualitas corpus: A curated collection of Java code for empirical studies," in *Asia Pacific Software Engineering Conference*, 2010.

[13] S. Blom, M. Huisman, and J. Kiniry, "How do developers use APIs? A case study in concurrency," in *International Conference on Engineering of Complex Computer Systems (ICECCS 2013)*. IEEE Computer Society Conference Publishing Services (CPS), 2013, pp. 212–221.

[14] Verified synchronizer specifications. [Online]. Available: http://fmt.ewi.utwente.nl/redmine/projects/vercors-verifier/wiki/synchronizers

[15] C. Haack, M. Huisman, C. Hurlin, and A. Amighi, "Permission-based separation logic for Java," submitted.

[16] M. Parkinson and G. Bierman, "Separation logic, abstraction and inheritance," in *Principles of programming languages (POPL '08)*. ACM, 2008, pp. 75–86.

[17] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll, "An overview of JML tools and applications," *STTT*, vol. 7, no. 3, pp. 212–232, 2005.

[18] D. Lea, "A Java fork/join framework," in *Proceedings of the ACM 2000 conference on Java Grande, JAVA '00*. ACM, 2000, pp. 36–43.

[19] ——, "The java.util.concurrent synchronizer framework," *Science of Computer Programming*, vol. 58, no. 3, pp. 293–309, Dec. 2005.

[20] A. Amighi, S. Blom, and M. Huisman, "Resource protection using atomics: Patterns and verifications," http://eprints.eemcs.utwente.nl/23306/, CTIT, University of Twente, Technical Report TR-CTIT-13-10, 2013.

[21] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv, "Local reasoning for storable locks and threads," in *5th Asian Conference on Programming languages and Systems*, ser. LNCS, vol. 4807. Springer, 2007, pp. 19–37.

[22] A. Hobor and C. Gherghina, "Barriers in concurrent separation logic," in *20th European Symposium on Programming (ESOP)*, ser. LNCS, G. Barthe, Ed., vol. 6602. Springer, 2011, pp. 276–296.

[23] S. Heule, K. R. M. Leino, P. Müller, and A. J. Summers, "Abstract read permissions: Fractional permissions without the fractions," in *Verification, Model Checking, and Abstract Interpretation*, ser. LNCS, R. Giacobazzi, J. Berdine, and I. Mastroeni, Eds., vol. 7737. Springer, 2013, pp. 315–334.

[24] R. Dockins, A. Hobor, and A. W. Appel, "A fresh look at separation algebras and share accounting," in *7th Asian Symposium on Programming Languages and Systems*, ser. LNCS, Z. Hu, Ed., vol. 5904. Springer, 2009, pp. 161–177.