Formal Specification of LinkedBlockingQueue Using Concurrent Separation Logic

Jeroen Meijer
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
j.j.g.meijer@student.utwente.nl

ABSTRACT

Proving the correctness of multi-threaded programs is a challenge. To meet this challenge Hurlin recently designed a method based on separation logic to specify concurrent Java-like programs with fork, join and re-entrant locks. In this study we evaluate the usability of Hurlin's method. This is done by developing a formal specification for Java's library class LinkedBlockingQueue and arguing why LinkedBlockingQueue respects this specification. In our project we also inspect the Java Synchronizer Framework. We conclude that Hurlin's program logic is very useful for specifying a Java library class, however we need to introduce some additional language constructs in order to be able to specify LinkedBlockingQueue's safety properties. We are able to specify if threads have permission to change the head and tail of the queue. To be able to specify the blocking properties of LinkedBlockingQueue we need to introduce a new specification formula *spec_lock* to specify which Lock protects a ConditionObject in Java's Synchronizer Framework.

1. INTRODUCTION

Since multi-threaded programs become more popular everyday their safety properties have to be guaranteed by a program specification. A program specification is a description of a program's behaviour. If an implementation respects this specification, we say the implementation is correct. The Java Modeling Language (JML) is a method widely used for showing the correctness of sequential Java programs. Showing the correctness of concurrent programs however, is much more of a challenge. Hurlin introduces a method for showing the correctness of multi-threaded Java-like programs in his PhD thesis [9]. Hurlin's method to reason about concurrent programs is a huge step forward towards a practically usable program logic. This thesis focuses on the soundness of the method rather than on its practical use. Therefore we will evaluate Hurlin's method in a practical example. We provide a specification for Java's concurrent library class LinkedBlockingQueue together with an argumentation of why the implementation of LinkedBlockingQueue respects this specification. During our study we found it very useful to combine JML with Hurlin's program logic. There is however not a formal syntax on how these languages are combined. Defining such a formal syntax is not part of this study due to time constraints. The VerCors project [5] – which is supported by an ERC starting grant [2] – will address this issue, see also Section 4 for related work. The result of this project is a specification for LinkedBlockingQueue. We use this specification to show to what extent it is possible to specify LinkedBlockingQueue, not to show that LinkedBlockingQueue is correct, we assume LinkedBlockingQueue is correct.

15th Twente Student Conference on IT June 20th, 2011, Enschede, The Netherlands.

2. BACKGROUND INFORMATION

A program specification can be used to show the correctness of a program's behaviour. For sequential programs there is a widely known method, namely the Java Modeling Language (JML). There has been at least one attempt [14, 3] to add multi-threading functionality to JML, but this study has not advanced enough to be used in a practical example. In a multi-threaded program, multiple threads have the ability to write to a memory location concurrently. These – often unwanted – data races can be guaranteed not to occur by using *separation logic* as a basis for a program specification. In this study we use such a program specification method for the Java library class *LinkedBlockingQueue*.

2.1 Concurrent separation logic

Concurrent separation logic (CSL) is an extension of Hoare logic which logically enforces correct synchronization of heap access [7]. CSL was introduced by O'Hearn [12] in the year 2007. Separation logic has the ability to reason about the contents of the memory. The heap – opposed to a stack – is shared between threads (within one JVM). Memory locations in a heap can be referred to with pointers and thus shared between threads. Separation logic comes with operations to manipulate and compare parts of the heap. These operations allow the heap to be split into disjoint parts in the specification. Reasoning about disjoint parts of the memory allows one not only to determine which parts are shared locations and which are not, it also allows one to determine which pointers are aliases [6]. Aliases are object references – or variables – that point to the same location on the heap.

The most important features of separation logic are access tickets and local reasoning. $x.f \mapsto v$ is called the *points-to* predicate and has a dual meaning. Firstly it asserts that object field $x.f$ contains the value v. Secondly it represents a permission to access the field $x.f$[7]. Local reasoning is supported by the operator $*$ and is called the *resource conjunction*. The formula $F * G$ means that the formula F holds for a part of the heap, while formula G holds for a different part of the heap. The following example

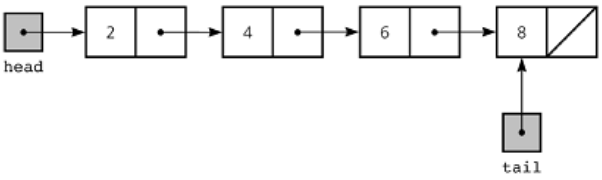

Figure 1. Illustration of a linked list

illustrates the use of the points-to predicate.

$$\{x.f \mapsto 1\}x.f = x.f + 4\{x.f \mapsto 5\}$$

In the example above the precondition not only indicates that $x.f$ has the value 1, it also expresses ownership of $x.f$. The program assigns $x.f+4$ to $x.f$. The postcondition indicates that $x.f$ has the value 5 and it grants a *ticket* to access $x.f$. The separation logic we use in our study is described in Hurlin's PhD thesis. His thesis introduces a method to reason about a concurrent Java-like language containing specific Java constructs, such as fork, join and re-entrant locks. Hurlin introduced the ASCII representation of the points-to predicate *PointsTo(x.f,π,v)*. The predicate expresses ownership by means of an access ticket π. The access ticket must be a fraction $\frac{1}{2^n}$ in the interval $(0, 1]$. In the example above we have an access ticket $\pi = 1$ for $x.f$, because $\pi = 1$ grants write access to a class' field. Every access ticket smaller than 1 grants read access. Permissions can be exchanged when threads fork or join.

Two important specification formula's of Hurlin's research are *LockSet(S)*, *S contains e* and quantifiers. LockSet(S) can be used to introduce a multiset of locks (S) that a thread holds. This multiset can be empty. The expression *S contains e* is used to express that multiset S contains object e. Two quantifiers that we use in our research are *exists* (*ex*) and *for all* (*fa*). They are identical to the JML keywords \exists and \forall. Furthermore we use the JML keyword \old. Every expression inside the parentheses of \old() is evaluated in the pre-state.

2.2 LinkedBlockingQueue

In our study applied Hurlin's method on LinkedBlockingQueue. Michael et al. [11] recommend the class LinkedBlockingQueue to be used in a situation where dedicated multiprocessors run under high contention. LinkedBlockingQueue is a Java library class which contains two important properties. The first property is that the head and tail can be locked independently. The second property of LinkedBlockingQueue is that threads can block when the queue is empty or full. Figure 1 [1] is an illustration of a linked list with pointers to the *head* and *tail*. Remind that LinkedBlockingQueue uses *last* as a pointer to the tail of the queue.

In addition to this concept of a linked list, LinkedBlockingQueue provides methods to *put elements to the tail* of the queue or *take elements from the head* of the queue. Furthermore LinkedBlockingQueue uses a blocking mechanism which will be covered later on. In Java's implementation of LinkedBlockingQueue the tail pointer is called *last*. A *Node* is defined as an element together with a pointer to the *next* Node, hence it is linked. The class is blocking and it uses a two-lock concurrent algorithm to enqueue and dequeue elements. For both the locking and blocking concepts we can provide some safety properties which should hold during the execution of a program.

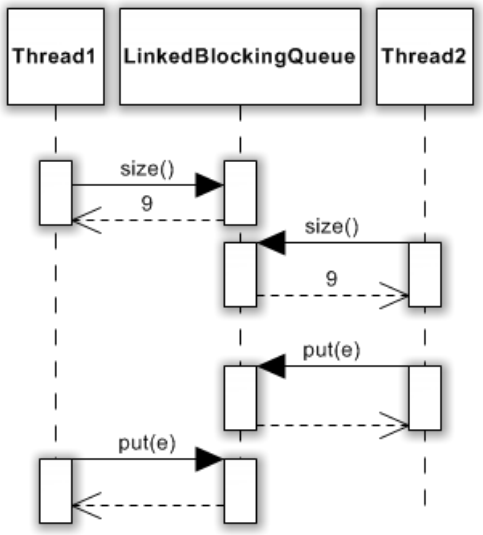

Figure 2. Illustration of a data race

Blocking algorithms allow a *slow or delayed process* to prevent faster processes from completing operations on a shared data structure indefinitely. When a blocking queue reaches its capacity a process which is trying to enqueue an element is blocked until another element is dequeued by another process. When a blocking queue has no elements a process trying to dequeue an element is blocked until another process enqueues an element. Two safety properties that should always hold for a blocking queue are as follows:

- no reads from an empty queue will occur,
- no writes into a queue which has reached its capacity occur.

In a LinkedBlockingQueue two locks are used to restrict access to both the tail and the head of the queue, thus making it possible to simultaneously enqueue and dequeue – two different – elements. The *head* lock disallows two elements from being concurrently removed from the queue, while the *tail* lock disallows two elements from being concurrently added to the queue. The two locks together prevent data races. A simplified situation of a data race is illustrated in Figure 2. Assume that for both threads it holds that the capacity of LinkedBlockingQueue is 10. This means that a thread may put an element into the queue if the size is less than 10. So if the threads are scheduled as illustrated a data race will occur, namely on getting the size of the queue. Access to the size of the queue should be synchronised.

For the two-lock algorithm [11] Michael et al. state that the following safety properties should hold:

- the linked list is always connected,
- nodes are only inserted after the last node in the linked list,
- nodes are only deleted from the beginning of the linked list,
- the pointer head always points to the first node in the linked list,

- the pointer tail always points to a node in the linked list.

In Section 6 one can find a formal specification of the afore mentioned seven safety properties.

2.3 The Implementation

LinkedBlockingQueue is available as a Java Library class since the introduction of Java Second Edition. In this project we specify the *constructors* and the methods *put* and *take*. One constructor of LinkedBlockingQueue initialises the queue with a default capacity of *Integer.MAX_VALUE*. The other constructor initialises the queue with a given capacity. As in each class that implements a Queue, LinkedBlockingQueue also implements the methods offer and remove. These both entail the same functionality as the methods put and take. However, offer and remove do not block if the queue reaches its capacity or become empty. These two methods are less suitable for specification, because we want to specify blocking properties with Hurlin's program logic. We therefore omit a specification for methods offer and remove.

2.4 The Synchronizer Framework

Many concurrent Java library classes such as LinkedBlockingQueue depend heavily on the *Java Synchronizer Framework* [10]. One essential class in this library is the *AbstractQueuedSynchronizer* class. This class is abstract so it is possible to write many different user implementations for this synchronizer. Also, there exist some standard subclasses of which one is *ReentrantLock* and is used in LinkedBlockingQueue. For a list of other synchronizer implementations we refer to the paper by Lea [10].

ReentrantLock – just like any other *Lock* – has one method *lock* to acquire the lock and one method to release the lock; *unlock*. Also one variant of method lock exist, namely *lockInterruptibly*, which throws an InterruptedException if the current thread is interrupted. For the purpose of blocking and unblocking of threads ReentrantLock employs a class named *ConditionObject* which implements the *Condition* interface. Lea [10] states the following.

A Condition object replaces the use of the Object monitor methods (e.g. Object.wait). a ConditionObject attached to a ReentrantLock acts in the same way as do built-in monitors (via Object.wait etc.), differing only in method names, extra functionality, and the fact that users can declare multiple conditions per lock.

So in our specification we are going to make use of the fact that every Lock is backed by a AbstractQueuedSynchronizer and every AbstractQueuedSynchronizer.ConditionObject by a Condition interface.

3. PROBLEM STATEMENT AND RESEARCH QUESTIONS

Hurlin's method to reason about concurrent programs is a huge step forward towards a practically usable program logic. The focus of Hurlin's work was on introducing and proving the soundness of a method, rather than on testing this method in practice. In the process towards an automated validation tool for concurrent programs, Hurlin's work has to be applied in practice to determine its usability. The following study questions will help us attaining this goal.

1. To what extent is it possible to specify a LinkedBlockingQueue with Hurlin's program logic?
2. To what extent is the specification of LinkedBlockingQueue useful?
3. Why does the implementation of LinkedBlockingQueue respect its specification?

4. RELATED WORK

The work closest related to Hurlin's – which will be used in our study – is from Parkinson [13]. Parkinson's thesis provides verification for Java-like programs with separation logic. Hurlin reuses most of Parkinson's ideas but differs on some points. *Middleweight Java* (MJ) together with an adaptation of separation logic is used for local reasoning. To the best of our knowledge no Java library classes were specified with MJ. The Bandera tool set [8] is a model checker to model-check properties of concurrent Java software.

Our study is a case study in which the work by Hurlin [7] is used in practice to specify a concurrent Java library class. We will not be the first to test Hurlin's work in practice. Burgman already compared [6] Hurlin's variant of separation logic with JML. Burgman provided a specification with separation logic and a specification with JML for a sequential mergesort and a concurrent mergesort algorithm for a linked list. He concluded that separation logic and JML both have their advantages and disadvantages and that they mainly support each other, but not replace each other. Current study done at the *VerCors* project (see [5]) focuses on integrating JML and separation logic.

5. METHOD OF RESEARCH

The study is done in two phases. In the first phase of the study we specified the class *LinkedBlockingQueue*. This class can be found in the Java package *java.util.concurrent*. Of the class LinkedBlockingQueue the most interesting methods are specified, namely put and take. Proving the correctness of the specification is omitted in this study due to time constraints. In the second phase we informally argue why the implementation of LinkedBlockingQueue respects this specification.

We have chosen to specify LinkedBlockingQueue, because it has some interesting safety properties to prove (see subsection 2.2). Furthermore LinkedBlockingQueue inherits interesting concepts like *re-entrant locks*, *conditions*.

6. THE SPECIFICATION

Remember that we want to specify five locking properties and two blocking properties, see Section 2.2. We divide the specification in two parts. The first part specifies the locking properties of the queue. The second part specifies the blocking properties of the queue. We provide inline code listings as fragments of LinkedBlockingQueue's specification but the complete specification can be found online [4]. In general, constructors of classes are specified with postconditions ensuring that locks are initialized and can be obtained. In this section we do not show how the constructors are specified, because in both Hurlin's and Burgman's thesis it has already been shown how constructors can be specified. Our specification of the constructors can also be found online. Along with the inline code listings we argue why LinkedBlockingQueue respects its specification.

6.1 The Locks

We first introduce an issue. Consider the last lines of code from the method *put*.

...

```
        putLock.unlock(); // Release the lock protecting the tail.
    }
    if (c == 0) { // c is the amount of elements before enqueueing
        .
        signalNotEmpty();
    }
} // End of put().
```

Between the execution of the line where a thread unlocks putLock and the line where the put method returns, other threads can also put elements in the LinkedBlockingQueue. Similarly for the method take:

```
    ...
        takeLock.unlock(); //Release the lock protecting the head.
    }
    if (c == capacity) { // c is the amount of elements before
          enqueueing and capacity is the capacity of the queue.
        signalNotFull();
    }
    return x;
} // End of take().
```

Between the execution of the line where a thread unlocks takeLock and the line were the take method returns, other threads can also take elements from the LinkedBlockingQueue. Because of these two similar issues we can only specify a short postcondition for both methods. For the method put the contract looks as follows:

```
/*
 *@ requires LockSet(S) * putLock.initialized
 *@ ensures \old(last) != last
 */
public void put(E e) throws InterruptedException
```

The method requires a LockSet with an arbitrary set of elements S and that the putLock is initialized. The postcondition only guarantees that there is another Node at the tail of the queue then there was in the pre-state i.e., \old(last) != last. Similarly for the method take we define a contract.

```
/*
 * @requires LockSet(S) * takeLock.initialized
 * @ensures \old(head) != head
 */
public E take() throws InterruptedException
```

Again we require a LockSet with an arbitrary set of elements S and that the takeLock is initialized. The postcondition can only guarantee that there is a different Node at the head of the queue then there was in the pre-state. Note that in both cases a thread does not need permissions on the tail or head to obtain such postconditions. The idea of this postcondition is obtained from the work of Vafeiadis et al. [15]. To strengthen our contract for LinkedBlockingQueue we use an invariant and a constraint. Invariants are properties that have to hold in all visible states. A constraint is used to constrain the way values change over time. However, for LinkedBlockingQueue, it is the case that properties about the head and tail can only be guaranteed if a thread has permission to access them. We have specified the following invariant for LinkedBlockingQueue. (Note that we use numbers to identify parts of the invariant and later on also for the constraint.)

```
/*
 *@ invariant perm(head, 1) * perm(last, 1) −*
 * (1) ((fa Node m . m.next != head)  *
 * (2) (ex Node n . n == last)   *
 * (3) (fa Node o . ex Node p . o.next == p);
 */
```

Each part of the invariant is true if a thread holds the putLock and the tailLock. Part *(1)* ensures "The pointer head always points to the first node in the linked list." We ensure this the other way around. We say that if for all Nodes its next Node does not point to the head Node then the head node is the first Node in the linked list. *(2)* ensures the safety property "The pointer tail always points to a node in the linked list." The third *(3)* part ensures that "the linked list is always connected." This holds even for the tail of the queue and therefore, for a LinkedBlockingQueue which size is less than two. This holds, because last.next always points to last. This is a design choice by the authors of LinkedBlockingQueue and is useful, because it indicates the end of the queue.

Consider the constraint for LinkedBlockingQueue.

```
/*
 *@ constraint perm(head, 1) * perm(last, 1) −*
 * (4) head.retainOrderState<\old(last)>;
 */
```

We constructed a state predicate in the class Node which is ensured by LinkedBlockingQueue's constraint.

```
/*
 *@ public pred retainOrderState<Node oldLast> =
 * exOld Node n . (this == n && n != last) ==>
 * this.next == n.next && this.next.retainorderState<last>;
 */
```

Part *(4)* strengthens the two safety properties "Nodes are only inserted after the last node in the linked list" and "Nodes are only deleted from the beginning of the linked list". For the methods put and take we have only provided a postcondition which says that the head and the tail are different in the post-state. But we also want to guarantee that nodes remain in the same order after a put or a take. Or, in other words, every Node which exists in both the pre-state and post-state point to the same *next* element. The predicate *retainOrderState* is called on the head Node and then iterates over the next node until the last Node in the pre-state. While iterating over the Nodes it guarantees that a Node's next Node is equal to the same Node's next Node in the pre-state. Two major issues we have encountered while specifying with *retainOrderState* is that we have no way to reason about the pre-state of the entire *heap*. We use the quantifier *exOld* which is actually the same as \exists in JML or *ex* in Hurlin's program logic, but it allows us to reason about the pre-state of the heap. Furthermore in our definition of retainOrderState one must read *n.next* as a reference in which next points to an address in the pre-state of the heap. We have not found another more elegant way to specify the constraint for LinkedBlockingQueue.

6.2 Blocking

Specifying the blocking part of LinkedBlockingQueue is easier then specifying the locking part. In our specification we have to specify that a lock must be held by a thread in order to block. This means that we have to introduce a specification formula which does not already exist in either Hurlin's program logic or JML. Every *Lock* uses a method *newCondition* to obtain a new Condition. A Condition can be used to block threads. Although LinkedBlockingQueue uses a ReentrantLock – which implements the interface Lock – we choose to specify Lock and not ReentrantLock so that every Lock – including ReentrantLock – inherits this specification. The specification formula we introduce is `spec_lock` L in which L is a Java variable of the type Lock. Consider the following specification for Condition.

```
//@ spec_lock lock
public interface Condition
```

This specifies that a new Condition object must have a Lock object to which it belongs. The specification for the interface Lock.newCondition() is as follows.

```
/*@ spec_lock this */ Condition newCondition();
```

Note that we use `spec_lock` twice, first as a declaration and second as an invocation. The above specification states that the Condition returned by the method newCondition has a Lock which is an instance of *this*. A thread can only obtain a lock if it invokes the method Lock.lockInterruptibly. Therefore the postcondition ensures a thread holds the lock.

```
//@ requires LockSet(S)
//@ ensures LockSet(S . this);
void lockInterruptibly() throws InterruptedException;
```

Note that lockInterruptibly() returns immediately.

Finally the contract for the method Condition.await looks as follows.

```
//@ requires LockSet(S) * S contains lock;
//@ ensures LockSet(S) * S contains lock;
void await() throws InterruptedException;
```

Clearly the specification of LinkedBlockingQueue is incorrect if a Lock is not held before calling Condition.await() i.e. Lock.lock() is not invoked before calling Condition.await().

7. FUTURE WORK

As mentioned in Section 2.4, LinkedBlockingQueue requires an abstract class and an interface of the Java synchronizer framework. It requires the abstract class AbstractQueuedSynchronizer and the interface Condition for obtaining locks and blocking threads. LinkedBlockingQueue requires method contracts from both AbstractQueuedSynchronizer and Condition, namely for the methods unlock, lockInteruptably and await. A useful topic for future research is to specify these two classes, because it can – like this study – evaluate Hurlin's program logic and provide a contract for two important classes within the Java synchronizer framework. Depending on the available time one could also provide contracts for the classes that implement AbstractQueuedSynchronizer and Condition like in ReentrantLock and ConditionObject to further evaluate Hurlin's program logic. We have already shown how one could specify which lock must be held in order to block a thread with Condition.await(), but it has to be defined formally.

In our study we have not investigated how the specification should deal with exceptions. One can imagine that an Exception might trigger different behaviour such that a lock is still held or not obtained. study has to be done on how JML's `signals` clause can be extended to deal with concurrent separation logic.

8. CONCLUSION

In this paper we have shown how Hurlin's program logic can be applied to Java's library class *LinkedBlockingQueue* with some additional constructs. So, to a large extent we were able to specify LinkedBlockingQueue with Hurlin's program logic. One hard aspect of applying separation logic for a contract for a Java class is that one has to carefully examine where an instance variable can be altered by a thread when a lock guarding this instance variable is not held in the body of a method. The contract of method put (see Section 6) only says the instance variable *last* is changed with respect to the pre-state. It does not say how it has changed and what happened to all other nodes in the LinkedBlockingQueue. We therefore conclude that the focus of programming by contract moves from method contracts to constraints and invariants. As a result it is mostly possible to ensure something about a class' state and only when a thread holds the necessary locks. A method contract is not relevant if it in the post-state does not hold a lock required for the postcondition.

Burgman concludes in his work that JML and SL support each other. We agree with his conclusion, but furthermore, we think they should be combined. Separation logic is a very good addition to the validation of Java code. For our study we found the use of \old and \result very useful. Those variables already exists in JML and the VerCors project should think about how to use them in their automated validation tool.

A specification formula has to be introduced much like the \old variable to make it possible to reason about the pre-state of the *heap*. Also, a minor note is that some constructs like exists written as 'ex T α' in Hurlin's Java like language already exist in JML. They should be combined somehow. Another minor issue is that the character for resource conjunction '*' is already reserved for JavaDoc.

In short we have found Hurlin's program logic in combination with JML very useful for specifying LinkedBlockingQueue, but we found the following important issues.

- To the best of our knowledge a specification in separation logic does not exist for Java's synchronizer framework. We have only provided a short specification, it has to be completed for the entire synchronizer framework.
- Most postconditions can only be guaranteed if the necessary locks are held, therefore invariants and constraints are far more useful then method contracts.
- With neither Hurlin's program logic, nor with JML it is possible to reason about the pre-state of the heap.
- We have shown a way to specify which lock must be held in order to block a thread, but a formal definition for this has to be developed.

The specification of LinkedBlockingQueue is useful in the sense that it shows that Hurlin's program logic is an essential step towards the verification of concurrent Java programs.

9. REFERENCES

[1] Linked Lists. https://wiki.cs.auckland.ac.nz-/compsci105ss/images/8/8c/Tail-linked-list.PNG, 2008. [Online; accessed 04/22/2011].

[2] ERC Starting Independent Researcher Grant. http://erc.europa.eu/index.cfm?-fuseaction=page.display&topicID=65, 2011. [Online; accessed 06/05/2011].

[3] JML Level C. http://www.eecs.ucf.edu/~leavens/-JML/jmlrefman/jmlrefman_2.html#SEC22, 2011. [Online; accessed 06/06/2011].

[4] LinkedBlockingQueue Specification. http://home.student.utwente.nl/j.j.g.meijer/-bachelorthesis, 2011. [Online; accessed 06/06/2011].

[5] VerCors project. http://fmt.ewi.utwente.nl/-projects/VerCors, 2011. [Online; accessed 06/06/2011].

[6] R. Burgman. Specifying Multi-Threaded Java Programs: A Comparison Between JML And Separation Logic. In *12th Twente Student Conference on IT*, 2010.

[7] C. Haack, M. Huisman, and C. Hurlin. Permission-Based Separation Logic for Multithreaded Java Programs. 2011. Manuscript.

[8] J. Hatcliff and M. Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. *CONCUR 2001–Concurrency Theory*, pages 39–58, 2001.

[9] C. Hurlin. *Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic*. PhD thesis, Université de Nice Sophia Antipolis, 2009.

[10] D. Lea. The java. util. concurrent synchronizer framework. *Science of Computer Programming*, 58(3):293–309, 2005.

[11] M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM, 1996.

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

[13] M. Parkinson. Local reasoning for Java. *PhD in Computer Science, University of Cambridge*, 2005.

[14] E. Rodriguez, M. Dwyer, C. Flanagan, J. Hatcliff, and G. Leavens. Extending JML for modular specification and verification of multi-threaded programs. *ECOOP 2005-Object-Oriented Programming*, pages 551–576, 2005.

[15] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. *CONCUR 2007–Concurrency Theory*, pages 256–271, 2007.