Proving Correctness of Highly-Concurrent Linearisable Objects

Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, Marc Shapiro

Proving Correctness of Highly-Concurrent Linearisable Objects

Viktor Vafeiadis
Computer Laboratory,
University of Cambridge
Viktor.Vafeiadis@cl.cam.ac.uk

Maurice Herlihy
Computer Science Dept.,
Brown University
mph@cs.brown.edu

Tony Hoare
Microsoft Research Cambridge
thoare@microsoft.com

Marc Shapiro
INRIA Rocquencourt & LIP6
http://www-sor.inria.fr/∼shapiro/

Abstract

We study a family of implementations for linked lists using *fine-grain synchronisation*. This approach enables greater concurrency, but correctness is a greater challenge than for classical, coarse-grain synchronisation. Our examples are demonstrative of common design patterns such as lock coupling, optimistic, and lazy synchronisation. Although they are are highly concurrent, we prove that they are linearisable, safe, and they correctly implement a high-level abstraction. Our proofs illustrate the power and applicability of *rely-guarantee* reasoning, as well of some of its limitations. The examples of the paper establish a benchmark challenge for other reasoning techniques.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Algorithms, verification

Keywords Concurrent programming, shared-memory concurrency, formal verification, linearisability, Rely-Guarantee reasoning.

1. Introduction

A *concurrent object* is a data structure (such as a list or a hash table) shared by multiple threads in a shared-memory multiprocessor. Classical implementations use *coarse-grain* synchronisation: the objects manipulated by the program are controlled by a single owner thread for as long as the program takes actions that might impact the program's or the object's invariants. Common design patters are monitors or a set of `synchronized` methods in Java™, which ensure that only one method call at a time can access the data structure. This approach makes it relatively easy to reason about correctness, but it limits concurrency, negating some of the advantages of modern multi-core or multi-processor architectures.

PPoPP'06 March 29–31, 2006, New York, New York, USA.

A *fine-grain* implementation permits more concurrency by permitting multiple threads to access inside a same object simultaneously. Reasoning about such algorithms is a greater challenge. Our contribution is to show, by extended example, that *rely-guarantee* reasoning [13] can be applied successfully to several rather challenging concurrent object implementation. We provide the first formal proof [22] of these algorithms, whereby we show that, despite high concurrency: (*i*) each operation appears to take effect instantaneously (a property called *linearisability* [11]); (*ii*) the low-level list code implements a high-level specification, that of a set; and (*iii*) the code is *safe*, i.e., each operation satisfies a specified post-condition and maintains the structural invariants of the object. Another contribution is our development of a version of *rely-guarantee* reasoning suitable for linearisable specifications.

There is a large body of research investigating fine-grained synchronisation [2, 8, 18, 19]. A number of standard design patterns have emerged. For example, in *lock coupling* [2], locks are acquired and released in a "hand-over-hand" order, acquiring the next lock in a sequence before releasing the previous. In *optimistic* synchronisation, a thread searches a data structure without acquiring locks, locks the sought-after component, and then validates after the fact (but before updating it) that the locked component is the correct one. In *lazy* synchronisation, the task of removing a component from a data structure is split into two phases: the component is *logically removed* simply by setting a flag, and later, the component can be *physically removed* as a "benevolent side-effect" of a later (or concurrent) method call. A *lock-free* implementation ensures that some thread always completes a method call in a finite number of steps, even in the presence of failures or delays by other threads. These techniques are important in practice: for example, the widely-used Java `util.concurrent` library includes fine-grain, lock-free implementations of lists and hash maps [16]. We suggest that the examples considered here be used as a kind of benchmark challenge for techniques for reasoning about highly-concurrent data structures.

We consider examples of list implementations that employ many of these fine-grained synchronisation techniques. We study in some detail a highly-concurrent linked-list implementation of a set [9] that uses a combination of optimistic and lazy synchronisation.

The rest of the paper proceeds as follows. We first introduce rely-guarantee reasoning and some notation, in §2. Then, Section 3 provides the axioms for a simple concurrent programming language. We examine a simple example, mutex locks, in §4. Section 5 discusses linearisation points. Now we may study fine-grain list al-

gorithms, in §6, and lazy concurrent lists in §7. §8 compares this work to the state of the art. We conclude, in §9 with a summary of results. In the interest of legibility, this paper focuses on the high-level ideas; detailed proofs and technical discussions are relegated to a companion technical report [22].

2. Setting the scene

Operationally, a concurrent system consists of a collection of sequential threads that apply atomic (i.e., isolated) read and write actions to shared memory. Threads can be interleaved arbitrarily: each successive pair of actions issued by one thread may be separated by an arbitrary sequence of actions invoked by concurrent threads, a phenomenon called *interference*. We assume that the shared memory is sequentially consistent[1] (SC): all threads see the operations performed to shared memory in the same order.

At a higher level of abstraction, threads communicate by calling methods of shared objects. Each shared object has a *type*, which defines a set of possible values and a set of *methods* to modify and observe the object's state. An object's methods can be invoked by concurrent threads.

2.1 Sequential reasoning

It is useful to distinguish between the *abstract* type being implemented (for example, a set of integers), and the underlying *concrete* type (for example, a linked list). In the standard proof methodology for sequential objects [7], a *linking invariant* (sometimes called an abstraction or refinement map) describes how concrete values represent abstract values. Not every concrete value necessarily represents an abstract value: only values satisfying a particular *representation invariant* are valid representations. Each of the object's methods may rely on this invariant when called, and it must guarantee the invariant holds when it returns, but it is free to violate the invariant while the method call is in progress. Because the representation and linking invariants capture what each method needs to know about the others, an object's methods can be implemented and verified independently, a property sometimes called *compositionality*.

Classically, a single atomic action is specified by a pair of predicates $(p,\ q)$, where p is the pre-condition assumed to hold when the action starts, and q is the post-condition established if and when the action terminates. As the post-condition describes an update, we write it as a "two-state" predicate, relating the state of the store at the start (written $\overleftarrow{\sigma}$) with the state σ which it leaves on termination. Initial and final values of each variable x are similarly denoted by $\overleftarrow{x}$ and x.

We shall use relational notation to abbreviate operations on predicates of two states. Relational composition of predicates describes exactly the intended behaviour of the sequential composition of programs:

$$(P; Q)(\overleftarrow{\sigma}, \sigma) \stackrel{\text{def}}{=} \exists\tau.\ P(\overleftarrow{\sigma}, \tau) \wedge Q(\tau, \sigma)$$

The program that makes no change to the state is described exactly by

$$\mathrm{ID}(\overleftarrow{\sigma}, \sigma) \stackrel{\text{def}}{=} (\overleftarrow{\sigma} = \sigma).$$

[1] For simplicity, our proofs assume that the order of function calls and returns is also sequentially consistent. This is, however, not strictly necessary, because the only way of detecting a non-SC execution would be through shared memory which is SC. The only place where our programs could encounter non-SC behaviour is in the lazy `contains` method of Section 7; this can be avoided in Java by declaring the .next and .marked fields as `volatile`.

The familiar notation R^* describes any finite number of iterations of the program described by R. It is defined

$$R^* \stackrel{\text{def}}{=} \mathrm{ID} \vee R \vee (R; R) \vee (R; R; R) \vee \cdots$$

2.2 Ownership-based methodologies

Ownership-based methodologies [12, 20] generalise sequential reasoning to concurrent objects that synchronise via coarse-grained synchronisation. At most one thread at a time can own an object. While the object is unowned, it must satisfy its representation invariant. While it is owned, however, the owner is free to violate the invariant, as long as it restores the invariant before it relinquishes ownership.

Concurrent objects that rely on fine-grained synchronisation do not provide the same clear-cut distinction between "owned" and "unowned" states. Because multiple threads may access an object concurrently, perhaps interleaving atomic operations in complex ways, fine-grained synchronisation requires identifying invariants that hold all the time, not just during coarse-grained intervals. However invariants are not enough. The updates to shared memory are constrained to satisfy certain predicates; and because they describe updates, these must be predicates over two states.

2.3 Rely-guarantee reasoning

In *rely-guarantee* (R-G) reasoning, each thread is assigned a *rely* condition that characterises the interference that thread can tolerate from the other threads. In return, the thread is assigned a *guarantee* condition that characterises how that thread can interfere with the others. Proving the safety of a program requires proving that (1) if each thread's rely condition is satisfied, then that thread satisfies its guarantee condition, and (2) each thread's guarantee condition implies the others' rely conditions.

Specification of a fine-grain concurrent program requires four predicates: $(p,\ R,\ G,\ q)$. The predicates p and q are the pre-condition and post-condition, as described above, and they describe the behaviour of the thread as a whole, from the time it starts to the time it terminates (if it does). R and G summarise the properties of the individual atomic actions invoked by the environment (in the case of R) and the thread itself (in the case of G). They are two-state predicates, relating the state $\overleftarrow{\sigma}$ before each individual atomic action to σ, the one immediately after that action. The *rely condition* R bounds the interference the thread can tolerate from the environment, whereas the *guarantee condition* G bounds the interference that it can impose on the other threads.

We require that the pre-condition is always preserved by the rely condition (i.e., $\overleftarrow{p} \wedge R \Rightarrow p$), but we do not impose a similar requirement for the post-condition.

In the rely condition, we often want to specify that there are certain variables the environment does not update, or that if some condition is satisfied, the environment actions preserve it [3]. We introduce the following notation for these specifications.

$$\begin{aligned}
\mathrm{ID}(x) &\stackrel{\text{def}}{=} (\overleftarrow{x} = x) \\
\mathrm{ID}(P) &\stackrel{\text{def}}{=} (\overleftarrow{P} = P) \\
\mathrm{Preserve}(P) &\stackrel{\text{def}}{=} (\overleftarrow{P} \Rightarrow P)
\end{aligned}$$

For convenience in the post-condition and guarantee condition, we define $\mathsf{Mod}(X)$ to mean that only variables in the set X are modified by the action. We extend these notations to multiple variables and conditions.

3. Axioms

In this section, we will define the semantics of a simple concurrent programming language by means of axioms and proof rules. We

will write $C \models (p, R, G, q)$ for the judgement saying that the program C meets the R-G specification (p, R, G, q).

Atomic actions are denoted by enclosing a program in diamond brackets $\langle C \rangle$. The corresponding proof rule is:

$$\frac{\{p\}\ C\ \{q\} \text{ in Hoare-logic}}{C \models (p,\ \text{Preserve}(p),\ q \vee \text{ID},\ q)} \quad \text{(ATOM)}$$

The pre-condition and post-condition remain the sequential ones; this thread guarantees it either does q or nothing; and the other threads are required to preserve the precondition. The implementation of atomicity must ensure that this interference cannot take place within the diamond brackets, so the proof of correctness of the atomic region are unaffected by interference. We will show later how the programmer is responsible for ensuring that interference is harmless in-between the atomic actions.

We take the convention in this paper that the only atomic statements are individual memory reads and writes. In the interest of legibility, we omit diamond brackets.

3.1 Sequential composition

For simplicity, we define sequential composition for programs with identical rely and guarantee conditions. (These conditions can always be weakened or strengthened as discussed in Section 3.3.)

$$\frac{\begin{array}{ll} C_1 \models (p_1,\ R,\ G,\ q_1) & \\ C_2 \models (p_2,\ R,\ G,\ q_2) & q_1 \Rightarrow p_2 \end{array}}{C_1\ ;\ C_2 \models (p_1,\ R,\ G,\ (q_1; R^*; q_2))} \quad \text{(SEQ)}$$

A sequential composition has the same pre-condition p_1 as its first operand, and the same guarantee conditions as both its operands. For its proper execution, the pre-condition of the second operand must follow from the post-condition of the first. Since the entire program will tolerate the same interference R as both its operands except at the very transition between the two operands, its total action will be given by the composition of the actions of its components accounting for environment interference in between.

3.2 Parallel composition

When threads run concurrently, each thread must ensure that its atomic actions do not interfere with the other threads except as expected. It is therefore essential to prove that the guarantee condition of each thread implies the rely condition of the others. The total action of the program is given by the composition of the actions of the two threads in either order, allowing for environment interference $(R_1 \wedge R_2)^*$ in between. All pre-conditions and rely conditions must hold, but concurrent combination can guarantee only the disjunction of the separate guarantee conditions.

$$\frac{\begin{array}{ll} C_1 \models (p_1,\ R_1,\ G_1,\ q_1) & G_1 \Rightarrow R_2 \\ C_2 \models (p_2,\ R_2,\ G_2,\ q_2) & G_2 \Rightarrow R_1 \end{array}}{C_1 \parallel C_2 \models (p_1 \wedge p_2,\ R_1 \wedge R_2,\ G_1 \vee G_2,\ q)} \quad \text{(PAR)}$$

where $q = (q_1; (R_1 \wedge R_2)^*; q_2) \vee (q_2; (R_1 \wedge R_2)^*; q_1)$.

In plain English, and generalising to any number of threads, it means that proving the safety of a parallel program reduces to: (*i*) a sequential proof of the post-condition and guarantee condition of each individual thread, assuming its rely condition is true, combined with (*ii*) a pairwise proof that every other thread's guarantee condition implies this thread's rely condition.

3.3 Refinement

Programs and specifications can be compared with each other by the standard refinement ordering. A stronger specification is possibly more desirable but more difficult to meet. When developing a program from its specification, it is always valid to replace the specification by a stronger one. A specification is weakened by weakening its post-condition or its guarantee condition. Conversely, it is strengthened by weakening its assumptions.

$$\frac{\begin{array}{ll} p' \Rightarrow p & R' \Rightarrow R \\ G \Rightarrow G' & q \Rightarrow q' \end{array} \quad C \models (p,\ R,\ G,\ q)}{C \models (p', R', G', q')} \quad \text{(REFINE)}$$

3.4 Further detail

We sometimes introduce *abstract variables* to aid reasoning. These variables do not affect program flow or outputs, and can be accessed only in *abstract statements* that write only to abstract variables and are guaranteed to terminate. Since these abstract operations do not actually take place, we can always group them atomically with their preceding or following atomic statement in the control flow.

When a new object is first allocated, only the allocating thread can access that object's fields; we say the object is *private*. Once the reference is placed in a shared location, it becomes *public*.[2] Since rely and guarantee conditions should not mention private objects, we introduce the following notation to quantify over all public objects of type T.

$$\forall_T\ x.\ P(x) \stackrel{\text{def}}{=} \forall\, x.\ (x : T \wedge \mathsf{Public}(x)) \Rightarrow P(x)$$

It is often useful to assume that each thread has a unique identifier. Within a thread specification the notation self refers to the identifier of the current thread. (One aspect of compositionality of R-G is that thread identifiers are abstract, and the only ones of interest are self and non-self.) The parallel composition rule, however, does not know anything about the special meaning of self. Therefore before applying the rule, one needs to substitute the actual thread identifier at all occurrences of self in the specifications. Given some expression P, we denote by $P_{[\mathsf{self}\ :=\ t]}$ this substitution of self by the actual thread identifier t in P. For example the implication $G_1 \Rightarrow R_2$ will become:

$$G_{1\ [\mathsf{self}\ :=\ 1]} \Rightarrow R_{2\ [\mathsf{self}\ :=\ 2]}$$

4. Mutual exclusion locks

This section specifies and implements a locking primitive (mutex) used in the later concurrent list algorithms. It is a simple example of the theory described in the previous section.

Formally, a mutex L is just a variable that holds the thread identifier of its owner, or null when unowned. A mutex provides two operations with (abstract) implementations: (where $\langle b \rightarrow C \rangle$ is a conditional critical region)

$$\begin{array}{rcl} L.\text{lock}() & \stackrel{\text{def}}{=} & \langle L.\text{owner} = \mathsf{null} \rightarrow L.\text{owner} := \mathsf{self} \rangle \\ L.\text{unlock}() & \stackrel{\text{def}}{=} & \langle L.\text{owner} := \mathsf{null} \rangle \end{array}$$

By applying the critical region axiom and refinement, we can deduce:

$$\begin{array}{rl} L.\text{lock}() & \models\ (L.\text{owner} \neq \mathsf{self},\ R,\ G,\ L.\text{owner} = \mathsf{self}) \\ L.\text{unlock}() & \models\ (L.\text{owner} = \mathsf{self},\ R,\ G,\ L.\text{owner} \neq \mathsf{self}) \end{array}$$

[2] A precise specification of the public property is outside the scope of this paper.

where

$$R = \mathit{LockRely} = \mathrm{ID}(L.\mathrm{owner} = \mathsf{self})$$
$$G = \mathit{LockGuar} = (\forall i \notin \{\mathsf{self}, \mathsf{null}\}.\ \mathrm{ID}(L.\mathrm{owner} = i))$$

Since these are the only two operations that can modify $L.\mathrm{owner}$, all threads automatically guarantee $\mathit{LockGuar}$. Note the guarantee condition of one thread implies the rely condition of another thread. More formally, for all $i \neq j$

$$\mathit{LockGuar}_{[\mathsf{self} := i]} \Rightarrow \mathit{LockRely}_{[\mathsf{self} := j]}$$

A common use for these axioms is to establish R-G conditions of the following form (where P is some property that we want to hold when we hold the lock: for example, stating that some variables/conditions are preserved).

$$R = \mathit{LockRely} \wedge (L.\overleftarrow{\mathrm{owner}} = \mathsf{self} \Rightarrow P) \quad \text{and}$$
$$G = \mathit{LockGuar} \wedge (L.\overleftarrow{\mathrm{owner}} \neq \mathsf{self} \Rightarrow P)$$

This makes the "locking discipline" explicit.

5. Linearisation points

A module implementing an abstract data type contains a local shared state and a set of methods operating on that state. The external specification of the module mentions only abstract state variables, which are disjoint from the concrete state. A linking invariant can be defined which relates the two.

In the sequential case, to prove that the concrete methods are equivalent to their abstract counterparts, it is sufficient to embed the abstract operations within the concrete method implementations and show that the linking invariant Inv is preserved by each method.

In the concurrent case, we must also establish that the externally visible (i.e., the abstract) effect of each method takes place at some instant between the method's invocation and return, atomically with respect to other concurrent method calls. This property is known as *linearisability* [11]. It ensures that every concurrent execution history is equivalent to some sequential one that preserves the order of non-overlapping operations.

Usually, the linearisability of an algorithm is shown by identifying a *linearisation point* in the code. At that point, we embed the abstract implementation of the algorithm and prove that the linking invariant is preserved by all atomic actions of the code (i.e., the guarantee condition contains $\mathrm{Preserve}(\mathit{Inv})$).

Sometimes however, the position of the linearisation point cannot be identified, because it depends on future behaviour. There are two cases to consider: the linearisation point either simply depends on the future execution of the same thread, or it also depends on the future execution of other concurrent operations. The first case arises quite commonly in optimistic algorithms, but is relatively simple to deal with. We identify a set of candidate linearisation points such that the linearisation point of the algorithm is the last one encountered in the control flow.

The second case is much subtler: depending on the scheduling, the linearisation point may in fact be between actions of other threads. Clearly, reasoning about the existence of such linearisation points requires some knowledge about the actions of other concurrent threads and requires global reasoning. With our approach at least, the precise assumptions are documented in the rely condition, and are therefore enforced by the parallel composition rule.

Since any changes after the linearisation point are not visible externally, the post-condition can be "lifted" to the whole function. Consider for instance an abstract integer x that is implemented in some complex way; for instance x maintains an audit trail. The increment function inc appears atomic, but internal operations might still occur after the linearisation point; for instance the audit trail is trimmed if it reaches a certain size. Other methods might take effect in the interval between the linearisation point and when the inc operation returns, and modify the value of x. It would appear that this violates the post-condition of inc; but since the internal operations have no visible effect, it is as if the interference occurred immediately *after* inc returned. Thus we may attach the post-condition $x = \overleftarrow{x} + 1$ to the whole inc function.[3]

$$\begin{array}{lll}
\mathit{AbsContains}(e) : & < & \mathit{AbsResult} := e \in \mathit{Abs} \quad > \\
\mathit{AbsAdd}(e) : & < & \mathit{AbsResult} := e \notin \mathit{Abs}\ ; \\
 & & \mathit{Abs} := \mathit{Abs} \cup \{e\} \quad > \\
\mathit{AbsRemove}(e) : & < & \mathit{AbsResult} := e \in \mathit{Abs}\ ; \\
 & & \mathit{Abs} := \mathit{Abs} \setminus \{e\} \quad >
\end{array}$$

Figure 1. Abstract list operations

6. Fine-grained list algorithms

In the rest of the paper, we consider a few fine-grain algorithms whose correctness we have proved manually using R-G reasoning. All the algorithms are concurrent, linearisable implementations of the *set* abstract data type presented in Fig. 1. It consists of the operations $\mathit{contains}$, add and remove: add adds the item to the set and returns true if the item was absent, otherwise it leaves the set unchanged and returns false. The remove behaves symmetrically.

The concrete representation used in all the algorithms is a sorted linked list representation. The list has two sentinel nodes: Head with value $-\infty$ and Tail with value $+\infty$. Intermediate nodes are sorted in a strictly increasing order; thus, there are no duplicates. We assume all elements e to be added to or removed from the list are in the range $-\infty < e < +\infty$. Each node in the list is associated with a lock; a private method $\mathit{locate}(e)$ locks and returns the two adjacent list nodes whose values enclose e. For brevity, we assume there is only one set in existence. This section is just an overview; the following one examines a challenging algorithm in detail.

6.1 Pessimistic list

The first algorithm (see Fig. 2) is pessimistic in its concurrency management: it always locks a node before accessing it.[4] locate traverses the list using *lock coupling*: the lock on some node is not released until the next one is locked, somewhat like a person climbing a rope "hand-over-hand." Note that lock operations are staggered, not nested.

An element is added to the set by inserting it in the appropriate position, while holding the locks of its two adjacent nodes. It is removed by redirecting the previous node's pointer, while both the previous and the current node are locked. This ensures that deletions and insertions can happen concurrently in the same list.

In the following representation invariant $\mathit{ListInv}$, the predicate $\mathsf{noOwn}(\mathrm{n})$ is introduced to allow temporary violation of the list structure. The invariant specifies that (*i*) Head and Tail contain the infinity values, (*ii*) if a public node other than Tail is unlocked, it points to a valid next node, (*iii*) if two unlocked nodes follow each other in the list, then their values are in ascending order; and (*iv*) the abstract set Abs and the set of values of non-sentinel nodes reachable from Head are equal. The latter clause links the abstraction with the implementation. Locked nodes may be arbitrarily modified by the thread holding the lock, as long as it maintains the ab-

[3] A formal discussion of 'lifting' is given in the companion technical report [22, Section 6.5].

[4] Our pseudocode uses a Java-like notation, without its more complex features such as object orientation, interrupts, class loaders, etc.

```
locate(e) :
  pred := Head ;
  pred.lock() ;
  curr := pred.next ;
  curr.lock() ;
  while (curr.val < e) {
    pred.unlock() ;
    pred := curr ;
    curr := curr.next ;
    curr.lock()
  } ;
  return pred, curr
```

```
add(e) :
  n1, n3 := locate(e) ;
  if n3.val ≠ e then
    n2 := new Node(e) ;
    n2.next := n3 ;
    n1.next := n2   [*A] ;
    Result := true
  else
    Result := false   [*B]
  endif ;
  n1.unlock() ;
  n3.unlock() ;
  return Result
```

```
remove(e) :
  n1, n2 := locate(e) ;
  if n2.val = e then
    n3 := n2.next   [*C] ;
    n1.next := n3 ;
    Result := true
  else
    Result := false   [*D]
  endif ;
  n1.unlock() ;
  n2.unlock() ;
  return Result
```

Figure 2. Lock-coupling list algorithm

straction and re-establishes the invariant when it unlocks the node.

$$\mathsf{noOwn}(\mathrm{n}) \stackrel{\text{def}}{=} \mathrm{n.owner} = \mathsf{null}$$

$$\begin{aligned} \mathit{ListInv} \stackrel{\text{def}}{=}\ & \mathsf{Node}(\mathrm{Head}) \\ & \wedge\ \mathrm{Head.val} = -\infty \wedge \mathrm{Tail.val} = +\infty \\ & \wedge\ \forall_{\mathsf{Node}}\ \mathrm{n}.\ (\mathsf{noOwn}(\mathrm{n}) \wedge \mathrm{n.val} < +\infty) \Rightarrow \mathsf{Node}(\mathrm{n.next}) \\ & \wedge\ \forall_{\mathsf{Node}}\ \mathrm{n\,m}.\ (\mathsf{noOwn}(\mathrm{n,m}) \wedge \mathrm{n} \rightarrow \mathrm{m}) \Rightarrow \mathrm{n.val} < \mathrm{m.val} \\ & \wedge\ \mathit{Abs} = \{\mathrm{n.val} \mid \mathrm{Head} \rightarrow^* \mathrm{n} \wedge \mathrm{n.val} \neq \pm\infty\} \end{aligned}$$

Note that the node allocated in *add* is private until the assignment n1.next := n2; therefore, as explained earlier, there can be no interference in the fields of n2 until that point. We write $\mathsf{Node}(\mathrm{n})$ for the assertion that n is a valid public node. Furthermore, we write $\mathrm{n} \rightarrow \mathrm{m}$ for $\mathsf{Node}(\mathrm{n.next}) \wedge \mathrm{n.next} = \mathrm{m}$, and $\rightarrow^*$ for the reflexive and transitive closure of $\rightarrow$.

To prove linearisability, we embed the abstract implementations *AbsAdd*, *AbsRemove* at the points marked `*A, *B, *C, *D` in Fig. 2 and show that (*i*) the list invariant is preserved by all atomic statements, and (*ii*) the post-condition of *add* and *remove* is $\mathit{Result} = \mathit{AbsResult}$. This proves that the marked points are indeed linearisation points.

To do the proof, the following R-G conditions are maintained by every thread:

$$\begin{aligned} R \stackrel{\text{def}}{=}\ & \forall_{\mathsf{Node}}\ \mathrm{n}.\ \mathrm{Preserve}(\mathit{ListInv}) \wedge \mathrm{n}.\mathit{LockRely} \\ & \wedge\ \mathrm{n}.\overleftarrow{\mathsf{owner}} = \mathsf{self} \Rightarrow \mathrm{ID}(\mathrm{n.val},\ \mathrm{n.next},\ \mathrm{Head} \rightarrow^* \mathrm{n}) \\ G \stackrel{\text{def}}{=}\ & \forall_{\mathsf{Node}}\ \mathrm{n}.\ \mathrm{Preserve}(\mathit{ListInv}) \wedge \mathrm{n}.\mathit{LockGuar} \\ & \wedge\ \mathrm{n}.\overleftarrow{\mathsf{owner}} \neq \mathsf{self} \Rightarrow \mathrm{ID}(\mathrm{n.val},\ \mathrm{n.next},\ \mathrm{Head} \rightarrow^* \mathrm{n}) \end{aligned}$$

The rely condition specifies that the environment actions preserve the list invariant and use locks properly. Furthermore, if a thread locks a node, other threads cannot update its fields or remove it from the list.

6.2 Optimistic list

Now consider the same algorithm, but with a different implementation for $\mathit{locate}(e)$ (*add* and *remove* are unchanged from Fig. 2; *locate* is given in Fig. 4, and *validate* in Fig. 3). The new *locate* is optimistic: it traverses the list without taking any locks, then locks two candidate nodes, and re-traverses the list to check whether the nodes are still present in the list and adjacent. If either test fails, the nodes are unlocked and the algorithm is restarted.

In this case, we cannot apply an ownership-based argument. While one thread has locked part of the list and is updating it, another thread may optimistically traverse it. The success of the optimistic traversal clearly depends on some properties of locked nodes (e.g., that they point to valid next nodes).

The representation invariant is similar to the one for the pessimistic algorithm, with the exception that the properties hold about

```
validate(pred, curr) {
  succ := Head ;
  while (succ.val < e)
    succ := succ.next ;
  return succ = curr and pred.next = curr
}
```

Figure 3. Optimistic *validate* function

all nodes, not only unlocked ones:

$$\begin{aligned} \mathit{ListInv} \stackrel{\text{def}}{=}\ & \mathsf{Node}(\mathrm{Head}) \wedge \mathrm{Head.val} = -\infty \\ & \wedge\ \forall_{\mathsf{Node}}\ \mathrm{n}.\ \mathrm{n.val} < +\infty \Rightarrow \mathsf{Node}(\mathrm{n.next}) \\ & \wedge\ \forall_{\mathsf{Node}}\ \mathrm{n\,m}.\ \mathrm{n} \rightarrow \mathrm{m} \Rightarrow \mathrm{n.val} < \mathrm{m.val} \\ & \wedge\ \mathit{Abs} = \{\mathrm{n.val} \mid \mathrm{Head} \rightarrow^* \mathrm{n} \wedge \mathrm{n.val} \neq \pm\infty\} \end{aligned}$$

The R-G conditions specify that all atomic actions preserve the list invariant, use locks properly and do not change the value of public nodes. Furthermore, if a thread locks a node, other threads cannot update its .next pointer or make it unreachable by any node from which it was previously reachable. This last condition is necessary for proving that the re-traversal corresponds to validating that the element is still in the list. Note that the order of the two pointer assignments in the add function is important (whereas they could be swapped in the pessimistic version).

$$\begin{aligned} R \stackrel{\text{def}}{=}\ & \forall_{\mathsf{Node}}\ \mathrm{n\,m}.\ \mathrm{Preserve}(\mathit{ListInv}) \\ & \wedge\ \mathrm{n}.\mathit{LockRely} \wedge \mathrm{ID(n.val)} \\ & \wedge\ \mathrm{n}.\overleftarrow{\mathsf{owner}} = \mathsf{self} \Rightarrow \mathrm{ID(n.next)} \wedge \mathrm{Preserve}(\mathrm{m} \rightarrow^* \mathrm{n}) \\ G \stackrel{\text{def}}{=}\ & \forall_{\mathsf{Node}}\ \mathrm{n\,m}.\ \mathrm{Preserve}(\mathit{ListInv}) \\ & \wedge\ \mathrm{n}.\mathit{LockGuar} \wedge \mathrm{ID(n.val)} \\ & \wedge\ \mathrm{n}.\overleftarrow{\mathsf{owner}} \neq \mathsf{self} \Rightarrow \mathrm{ID(n.next)} \wedge \mathrm{Preserve}(\mathrm{m} \rightarrow^* \mathrm{n}) \end{aligned}$$

7. Lazy list

In this section, we study a highly concurrent implementation using optimistic and lazy techniques, due to Heller et al. [9], presented in Fig. 4. The concrete representation is the same as the one used by the algorithms in the previous section. In addition, however, nodes have a .marked flag, which is set when the node is deleted. The implementation of *contains* takes no locks.

An element is added as before. An element is removed in two stages: first, the node is *logically* removed by setting the .marked flag; then it is *physically* removed by redirecting reference fields. Concurrent membership tests traverse the list without checking the .marked flag. This flag is checked only when a candidate node is found. Similarly, *locate* ignores the flag while traversing the list. When the method locates and locks the two candidate nodes,

```
locate(e) :
 while (true) {
   pred := Head ;
   curr := pred.next ;
   while (curr.val < e) {
     pred := curr ;
     curr := curr.next
   } ;
   pred.lock() ;
   curr.lock() ;
   if validate(pred, curr) then
     return pred, curr
   else
     pred.unlock() ;
     curr.unlock()
 }
```

```
contains(e) :
 curr := Head ;
 while (curr.val < e)
   curr := curr.next ;
 if curr.marked then
   return false
 else
   return curr.val = e

validate(pred, curr) :
 if ¬pred.marked
    and ¬curr.marked
    and pred.next = curr then
     return true
 else
     return false
```

```
add(e) :
 same as lock-coupling

remove(e) :
 n1, n2 := locate(e) ;
 if n2.val = e then
   n2.marked := true   [*C] ;
   n3 := n2.next ;
   n1.next := n3 ;
   Result := true
 else
   Result := false   [*D]
 endif ;
 n1.unlock() ;
 n2.unlock() ;
 return Result
```

Figure 4. Lazy list algorithm

it *validates* them by checking they are adjacent and unmarked. If validation fails, the *locate* operation is restarted.

Because *contains* is completely wait-free, this algorithm crucially depends on global invariants, such as the list being sorted, which *must hold at all times*, even when part of the list is locked and local updates are performed.

We prove safety properties of the algorithm using the axioms, the R-G conditions for mutual exclusion locks, and reasoning based on linearisation points. The proofs, besides the linearisability of *contains*, are formal, but manual; thus they might contain errors. The linearisability of *contains* is treated informally.

Note that ownership-based reasoning is inadequate for this example. When a resource is locked for writing by one thread, its rely condition permits other threads to update it in certain restricted ways. For example, threads can scan through or remove list elements that are currently locked or marked for removal.

7.1 Representation Invariant

There are two sentinel nodes Head and Tail with values $\pm\infty$, which cannot be deleted. All nodes in the list are *public*, and public nodes other than Tail point to other public nodes; the nodes are sorted. In addition, public nodes that are not in the list (i.e., not reachable from the head of the list), are necessarily marked because the *remove* method first marks a node before removing it physically.

$$
\begin{aligned}
&\mathit{ListInv} \stackrel{\text{def}}{=} \\
&\quad \mathsf{Node}(\mathrm{Head}) \wedge \mathrm{Head.val} = -\infty \wedge \neg \mathrm{Head.marked} \\
&\wedge\ \mathsf{Node}(\mathrm{Tail}) \wedge \mathrm{Tail.val} = +\infty \wedge \neg \mathrm{Tail.marked} \\
&\wedge\ \forall_{\mathsf{Node}}\, \mathrm{n}.\ \mathrm{n.val} < +\infty \Rightarrow \mathsf{Node}(\mathrm{n.next}) \\
&\wedge\ \forall_{\mathsf{Node}}\, \mathrm{n}\,\mathrm{m}.\ \mathrm{n} \rightarrow \mathrm{m} \Rightarrow \mathrm{n.val} < \mathrm{m.val} \\
&\wedge\ \forall_{\mathsf{Node}}\, \mathrm{n}.\ \mathrm{Head} \rightarrow^* \mathrm{n} \vee \mathrm{n.marked} \\
&\wedge\ \mathit{Abs} = \{\mathrm{n.val} \mid \mathsf{Node}(\mathrm{n}) \wedge \neg \mathrm{n.marked} \wedge \mathrm{n.val} \neq \pm\infty\}
\end{aligned}
$$

The last line of the invariant defines the abstract variable Abs to be the set of values contained in non-marked public nodes other than Head and Tail.[5] This line is the coupling invariant that relates the abstract and the concrete states.

In contrast to methods based on coarse-grained ownership, the list invariant holds at all points during execution: it holds initially, and is preserved by the rely and guarantee conditions.

[5] From the previous line, these nodes are reachable from Head.

7.2 Pre-conditions and post-conditions

All the methods share the same pre-condition:

$$\mathit{Pre} \stackrel{\text{def}}{=} \mathit{ListInv} \wedge -\infty < e < +\infty$$

The post-conditions of the external methods *contains*, *add* and *remove* are just their abstract implementations given in Fig. 1 (conjoined with $\mathit{ListInv}$), whereas the post-condition of $(\mathrm{pred}, \mathrm{curr}) := \mathit{locate}(e)$ is given below.

$$
\begin{aligned}
\mathit{locate.Post} \stackrel{\text{def}}{=}\ & \mathit{ListInv} \wedge \mathrm{Head} \rightarrow^* \mathrm{pred} \rightarrow \mathrm{curr} \\
& \wedge\ \mathrm{pred.val} < e \leq \mathrm{curr.val} \\
& \wedge\ \mathrm{pred.owner} = \mathrm{curr.owner} = \mathsf{self} \\
& \wedge\ \neg \mathrm{pred.marked} \wedge \neg \mathrm{curr.marked}
\end{aligned}
$$

7.3 Rely and guarantee conditions

Each thread relies on the fact that its environment preserves the list invariant, obeys the locking rely condition, does not update fields of nodes locked by the thread, does not unmark deleted nodes, and does not change the values of public nodes.

$$
\begin{aligned}
R \stackrel{\text{def}}{=}\ & \forall_{\mathsf{Node}}\, \mathrm{n}\,\mathrm{m}.\ \mathrm{Preserve}(\mathit{ListInv}) \wedge \mathrm{n}.\mathit{LockRely} \\
& \wedge\ \mathrm{n}.\overleftarrow{\mathrm{owner}} = \mathsf{self} \Rightarrow \mathrm{ID}(\mathrm{n.next}, \mathrm{n.marked}) \\
& \wedge\ \mathrm{n}.\overleftarrow{\mathrm{owner}} = \mathsf{self} \Rightarrow \mathrm{Preserve}(\mathrm{Head} \rightarrow^* \mathrm{n}) \\
& \wedge\ \mathrm{Preserve}(\mathrm{n.marked}) \wedge \mathrm{ID}(\mathrm{n.val}) \\
& \wedge\ \mathrm{Preserve}(\mathrm{m} \rightarrow^* \mathrm{n} \vee \mathrm{n.marked})
\end{aligned}
$$

Hence, each thread also promises to preserve the list invariant, not to update fields n.next and n.marked unless it locks n, not to unmark marked nodes, and not to change n.val for a public node n.

$$
\begin{aligned}
G \stackrel{\text{def}}{=}\ & \forall_{\mathsf{Node}}\, \mathrm{n}\,\mathrm{m}.\ \mathrm{Preserve}(\mathit{ListInv}) \wedge \mathrm{n}.\mathit{LockGuar} \\
& \wedge\ \mathrm{n}.\overleftarrow{\mathrm{owner}} \neq \mathsf{self} \Rightarrow \mathrm{ID}(\mathrm{n.next}, \mathrm{n.marked}) \\
& \wedge\ \mathrm{n}.\overleftarrow{\mathrm{owner}} \neq \mathsf{self} \Rightarrow \mathrm{Preserve}(\mathrm{Head} \rightarrow^* \mathrm{n}) \\
& \wedge\ \mathrm{Preserve}(\mathrm{n.marked}) \wedge \mathrm{ID}(\mathrm{n.val}) \\
& \wedge\ \mathrm{Preserve}(\mathrm{m} \rightarrow^* \mathrm{n} \vee \mathrm{n.marked})
\end{aligned}
$$

7.4 Proof of safety

Locking ensures that the rely condition is implied by the guarantee condition of other threads.

$$G_{[\mathsf{self}\ :=\ i]} \Rightarrow R_{[\mathsf{self}\ :=\ j]} \ \text{ for all } i \neq j$$

We now need to prove that the guarantee condition holds for all atomic actions of the algorithm. (1) $\mathit{LockGuar}$ holds, because locking is performed using its prescribed interface. (2) n.next and

$\mathrm{n.marked}$ are updated only for private or locked nodes. (3) Only locked nodes are removed from the list ($\mathrm{n2}$ in *delete*). (4) For a public node n, $\mathrm{n.marked}$ is never set to false and there are no assignments to $\mathrm{n.val}$. (5) Only marked nodes can be physically removed from the list. Finally, most statements do not update the list, trivially preserving the list invariant $ListInv$. Here are the interesting exceptions:

- In add, $\mathrm{n1.next} := \mathrm{n2}$. The newly created node $\mathrm{n2}$ has already been fully initialised, $\mathrm{n1.val} < \mathrm{n2.val} = e$, and e is also added to Abs.
- In $remove$, $\mathrm{n2.marked} :=$ false. The element e is removed from the abstract set Abs.
- In $remove$, $\mathrm{n1.next} := \mathrm{n3}$. Node $\mathrm{n2}$ was already marked, and the marking is preserved by the guarantee condition. This node may therefore become unreachable from the head of the list.

The sequential proof of *locate* is straightforward, and relegated to the technical report [22]. For the public methods, it is straightforward to prove the post-condition $ListInv$, but this is not enough, because we must show that e was added to or removed from the list. By inlining the abstract implementation $AbsAdd$ and $AbsRemove$ at the points marked in the code of add and $remove$, we can show that post-condition $Result = AbsResult$. It follows that the points marked `*A, *B, *C, *D` in add and $remove$ are valid linearisation points, and so the abstract post-conditions can be lifted to become post-conditions of the whole add and $remove$ methods.

7.5 Linearisability of contains(e)

The linearisability of $contains$ is much subtler; the simple method above cannot prove that $contains$ is linearisable. In fact, the rely condition given in Section 7.3 allows certain problematic environments in which $contains$ is not linearisable. The difficulty arises because the post-condition $Result = AbsResult$ is invalid for any placement of $AbsContains$ in the body of $contains$. We will show, somewhat informally, that under an additional constraint (i.e., a condition conjoined to R and G) $contains$ is linearisable.

It is useful to consider separately the case when $contains(e)$ returns true and when it returns false. If it returns true, it is easy to show that the last assignment to curr in the loop and the read of the $\mathrm{.marked}$ bit are both valid linearisations points. (We show that $contains(e)$ can derive the post-condition $Result \Rightarrow AbsResult$.)

If, however, $contains(e)$ returns false, the proof is more subtle. Initially, when $contains(e)$ starts executing, either e is in the list or it is not. If e is not in the list, then clearly the linearisation point can be identified to be that initial point. If e was in the list, that means that there exists a (unique) unmarked node m that is reachable from Head and contains the value e.

If the node m never gets marked during the execution of $contains(e)$ then $contains(e)$ will find that node and return true. (We show that under the precondition $ListInv \wedge \mathrm{Head} \rightarrow^* \mathrm{m} \wedge \mathrm{m.val} = e \wedge \neg\mathrm{m.marked}$ and the rely condition $R \wedge \mathrm{ID(m.marked)}$, the postcondition $Result =$ true is derivable.) Since this contradicts with our assumption that $contains(e)$ returns false, we conclude that the node m must have been marked (and perhaps even physically removed).

This, however, does not mean that e will not be in the list at the next scheduling of $contains(e)$; indeed, another node containing e could have been added in the meantime. What is important is that the marking and the addition of a new node cannot happen atomically. We can capture this additional requirement by the means of an additional predicate N to be conjoined to both the rely and guarantee conditions of all the methods.

$$N \stackrel{\text{def}}{=} \forall_{\mathsf{Node}}\, \mathrm{n}.\ (\neg\overleftarrow{\mathrm{n.marked}} \wedge \mathrm{n.marked}) \Rightarrow \mathrm{n.val} \notin Abs$$

This is a genuine two-state predicate: immediately after the step when the $\mathrm{.marked}$ bit is set, n is not in the abstract set. This guarantees that a linearisation point exists; it is exactly after the $\mathrm{.marked}$ bit was set. Further, note that N is not transitive and hence $R \neq R^*$. The proof that all atomic actions satisfy N is simple.

8. Previous work

Owicki and Gries originally identified the concept of non-interference [21]. Their proof method was extended by Cliff Jones, who invented the Rely-Guarantee Methodology [13]. As the parallel composition rule appears to be based on circular reasoning, Abadi and Lamport [1] studied whether it is sound. They conclude that it is sound for safety conditions, and provide a condition ensuring soundness in the case of liveness as well. Our work has only considered safety.

Elsewhere [15] Lamport argues that R-G only makes proofs harder, by imposing a predefined modular structure to a proof that is necessarily global. We argue that the compositional nature of R-G constitutes a powerful reasoning tool (see Conclusion). However, some problems, such as the identification of the linearisation point in Section 7.5, can be tackled more directly by global reasoning. Further research would be needed to combine the advantages of both techniques.

In Dingel's notation [3] rely (resp. guarantee) conditions are predicates that are preserved by the environment's (resp. the program's) atomic actions. This is a powerful idea, which we re-use in our $\mathrm{ID}(p)$ and $\mathrm{Preserve}(p)$ notations. For Dingel a condition is a single-state predicate; however the additional expressivity of two-state predicates is important for some examples, e.g., Section 7.5.

We rely on annotating the original Java program with a specification. We expect that the our methodology can also be applied within languages such as TLA [14] or IO Automata [17]. These have been designed for reasoning about concurrent algorithms and have been used successfully in proving properties such as safety, liveness, or compliance of implementation to specification. They have been used mainly for whole-system proofs of distributed algorithms.

The R-G approach is widely used in hardware verification (where it is called assume-guarantee). For instance, Henzinger et al. [10] verify that a complex pipe-line implements a high-level processor specification. Both are expressed in a reactive language; writes are synchronous and parallelism occurs as non-deterministic choice. Proofs are assisted by a formal verification tool.

The recent work of Flanagan et al. [5] is the most similar to ours. They use a prover to statically verify properties of legacy multithreaded Java programs (annotated with pre-conditions, post-conditions, invariants, and R-G conditions), using R-G and other methods to decompose the proof along procedure and thread boundaries. They provide a number of examples of bugs detected by their tool in sizable programs.

Our highly-concurrent linked list algorithms are similar to ones previously published [8, 9, 18, 19]. Doherty, Groves, Luchangco and Moir [4] offer a formal proof of the lock-free queue algorithm of Michael and Scott [19]. Their proof is machine-verified; ours remains manual, hence more prone to error (automated verification is future work). They model the specification and the implementation as automata, and prove that all executions of the latter are allowed by the former. In contrast, our approach proceeds directly from the source text, and we prove that the implementation conforms to the specification by embedding the latter in the text.

Recently, Gao, Groote and Hesselink studied a lock-free algorithm for hash tables [6]. They prove both safety and liveness using the PVS verifier, by equipping the program text, line by line, with a large number of assertions and invariants.

9. Conclusion

Modern research on concurrent data structures has increasingly focused on algorithms that use fine-grained synchronisation rather than coarse-grained locking. Reasoning about these fine-grained algorithms is more challenging, particularly since conventional proof techniques may not extend easily into this new domain. In this paper, we have shown how R-G reasoning can be applied to prove the correctness of several non-trivial shared-memory concurrent algorithms based on fine-grained synchronisation. We have proved both that they satisfy their invariants, and that they implement a high-level specification. The accompanying technical report [22] contains the detailed proofs. We think that this example, or similar ones, could be used to compare the expressive power of techniques for reasoning about highly-concurrent data structures.

In our experience, the R-G structure constitutes a powerful reasoning tool. It allows us to decompose the proof according to the modular structure of the program; one may reason about a single abstraction in isolation from others, using standard sequential techniques. For instance, we identified a crucial rely condition $\mathrm{Preserve}(\mathrm{m} \rightarrow^{*} \mathrm{n})$, which was never recognised previously. R-G conditions document precisely what interference is allowed between abstractions: thus, the "locking discipline" is formalised and verifiable, and it will be clear whether our list module remains correct in the context of a larger program.

So far, we have addressed only safety properties of these fine-grained concurrent algorithms. Reasoning about their liveness remains future work.

Acknowledgments

We thank Lindsay Groves, Leslie Lamport, Victor Luchangco, Mark Moir, and Alan Mycroft for their insightful comments and discussions. We acknowledge a scholarship from the Cambridge Gates Trust (Vafeiadis), and support from Microsoft Research Cambridge (Herlihy, Shapiro, Vafeiadis).

References

[1] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Prog. Lang. Syst.*, 17(3):507–534, May 1995.

[2] R. Bayer and M. Schkolnick. Concurrency of operations on B-Trees. *Acta Informatica*, 9:1–21, 1977.

[3] J. Dingel. Computer-assisted assume/guarantee reasoning with VeriSoft. In *Proc. Int. Conference on Software Engineering (ICSE-25)*, pages 138–148, Portland, Oregon, USA, May 2003. IEEE Computer.

[4] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *Int. Conf. on Formal Techniques for Networked and Dist. Sys. (FORTE 2004)*, volume 3235 of *Lecture Notes in Computer Science*, pages 97–114, Madrid, Spain, Sept. 2004. IFIP WG 6.1, Springer-Verlag.

[5] C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular verification of multithreaded programs. *Theoretical Computer Science*, 338(1–3):153–183, June 2005.

[6] H. Gao, J. F. Groote, and W. H. Hesselink. Lock-free dynamic hash tables with open addressing. *Distrib. Computing*, 18(1):21–42, July 2005.

[7] J. V. Guttag, E. Horowitz, and D. R. Musser. Abstract data types and software validation. *Commun. ACM*, 21(12):1048–1064, 1978.

[8] T. Harris. A pragmatic implementation of non-blocking linked-lists. *Lecture Notes in Computer Science*, 2180:300–314, 2001.

[9] S. Heller, M. Herlihy, V. Luchangco, M. Moir, B. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *9th International Conference on Principles of Distributed Systems (OPODIS)*, Dec. 2005.

[10] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Int. Conf. on Computer-Aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*, pages 440–451, Vancouver, BC, Canada, June 1998. Springer-Verlag.

[11] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3):463–492, July 1990.

[12] B. Jacobs, K. R. M. Leino, and W. Schulte. Verification of multithreaded object-oriented programs with invariants. In *W. on Specification and Verification of Component-Based Systems*, Oct. 2004.

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

[14] L. Lamport. The temporal logic of actions. *ACM Trans. Prog. Lang. Syst.*, 16(3):872–923, 1994.

[15] L. Lamport. Composition: A way to make proofs harder. In *Proc. COMPOS'97 Symp.*, volume 1536 of *Lecture Notes in Computer Science*, pages 402–423. Springer-Verlag, 1997.

[16] D. Lea. Concurrent hash map in JSR166 concurrency utilities. `http://gee.cs.oswego.edu/dl/concurrency-interest/`.

[17] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. Symp. on Principles of Distributed Computing*, pages 137–151, New York, NY, USA, 1987. ACM Press.

[18] M. Michael. High-performance dynamic lock-free hash tables and list-based sets. In *Symposium on Parallel Algorithms and Architectures*, pages 73–82. ACM Press, 2002.

[19] M. Michael and M. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[20] P. W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR*, pages 49–67, 2004.

[21] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, May 1976.

[22] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. A safety proof of a lazy concurrent list-based set implementation. Technical Report UCAM-CL-TR-659, University of Cambridge, Computer Laboratory, 2006.