



HAL
open science

Proving Correctness of Highly-Concurrent Linearisable Objects

Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, Marc Shapiro

► **To cite this version:**

Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, Marc Shapiro. Proving Correctness of Highly-Concurrent Linearisable Objects. [Research Report] RR-5716, INRIA. 2005, pp.20. inria-00070301

HAL Id: inria-00070301

<https://inria.hal.science/inria-00070301v1>

Submitted on 19 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

*Proving Correctness of Highly-Concurrent
Linearisable Objects*

Viktor Vafeiadis — Maurice Herlihy — Tony Hoare — Marc Shapiro

N° 5716

11 octobre 2005

Thème COM



*R*apport
de recherche



Proving Correctness of Highly-Concurrent Linearisable Objects

Viktor Vafeiadis^{*}, Maurice Herlihy[†], Tony Hoare[‡], Marc Shapiro[§]

Thème COM — Systèmes communicants
Projet Regal

Rapport de recherche n° 5716 — 11 octobre 2005 — 20 pages

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 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.

Key-words: concurrent programming, shared-memory concurrency, formal verification, linearisability, Rely-Guarantee reasoning

* Computer Laboratory, University of Cambridge

† Computer Science Dept., Brown University

‡ Microsoft Research Cambridge

§ INRIA Rocquencourt & LIP6

Prouver le caractère correct d'objets hautement concurrents linéarisables

Résumé : Nous étudions une famille de mises en œuvre de listes chaînées utilisant de la *synchronisation à grain fin*. Cette approche permet plus de parallélisme, mais il est plus difficile de prouver son caractère correct qu'avec de la synchronisation classique à gros grain. Nos exemples utilisent des schémas de conception comme les verrous couplés, la synchronisation optimiste, et la synchronisation paresseuse. Bien qu'ils permettent un haut degré de concurrence, nous prouvons qu'ils sont linéarisables et sûrs, et qu'ils mettent correctement en œuvre une abstraction de haut niveau. Ces preuves illustrent bien la puissance et l'applicabilité du raisonnement «Rely-Guarantee» ainsi que certaines de ses limitations. Les exemples de ce papier peuvent être vus comme un défi aux autres techniques de preuve.

Mots-clés : programmation parallèle, parallélisme en mémoire partagée, vérification formelle, linéarisabilité, raisonnement «Rely-Guarantee»

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. By this we mean that 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. Typical design patterns for coarse-grain synchronisation include monitors [11] or a set of `synchronised` 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.

A *fine-grain* implementation permits more concurrency; for instance it may allow permit 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 of these algorithms, whereby we show that, despite high concurrency: (i) each operation appears to take effect instantaneously (a property called *linearisability* [10]); (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, 7, 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 go in some detail into a highly-concurrent linked-list implementation of a set [8] that uses a combination of optimistic and lazy synchronisation.

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.¹ 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*.

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 [6], 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 $\overline{\sigma}$) with the state σ which it leaves on termination. Initial and final values of each variable x are similarly denoted by \overline{x} and x .

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.

¹For simplicity, our proofs assume sequential consistency (SC). The only place where our programs could encounter non-SC behavior is in the lazy `contains` method of Section 7; this can be avoided in Java by declaring the `.next` and `.marked` fields as `volatile`.

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.

2.4 Notation

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 sequential 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

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

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

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

It describes interference by an unknown number of atomic actions from other threads.

$$\frac{\begin{array}{ccc} p_1; R_1 \Rightarrow p_1 & R_1; q_1; R_1 \Rightarrow q_1 & G_1 \Rightarrow R_2 \\ p_2; R_2 \Rightarrow p_2 & R_2; q_2; R_2 \Rightarrow q_2 & G_2 \Rightarrow R_1 \end{array}}{(p_1, R_1, G_1, q_1) \parallel (p_2, R_2, G_2, q_2) = (p_1 \wedge p_2, R_1 \wedge R_2, G_1 \vee G_2, q_1 \wedge q_2)}$$

Figure 1: Parallel composition rule

Programs and specifications can be compared with each other by the standard refinement ordering. The notation $A \sqsupseteq B$ means that A is a stronger specification than B , 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.

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

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.

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

For convenience in the post-condition and guarantee condition, we define $\text{Mod}(X)$ 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 adopt the convention that a program is represented by the strongest specification that it can be proved to meet.

Atomic actions are denoted by enclosing a program in diamond brackets $\langle C \rangle$. As a sequential program, C can be modelled as a predicate pair (p, q) . As an atomic action, it becomes:

$$\langle (p, q) \rangle \stackrel{\text{def}}{=} (p, \text{true}, q, q)$$

The pre-condition and post-condition are unchanged. The guarantee condition is just the post-condition, and the rely condition allows arbitrary concurrent behaviour. 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 the only atomic statements are individual memory reads and writes; atomicity is guaranteed by the hardware with no overhead. We are therefore justified, in the interests of legibility, to 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 above.)

$$\frac{q_1; R^* \Rightarrow p_2}{(p_1, R, G, q_1); (p_2, R, G, q_2) = (p_1, R, G, (q_1; R^*; q_2))}$$

A sequential composition has the same pre-condition p_1 as its first operand, and the same guarantee conditions as both its operands. Except at the very transition between the two operands, it will tolerate the same interference R as both its operands. To deal with interference at the transition point, an extra proof obligation is imposed: the first operand's post-condition must imply the second operand's pre-condition despite interference by the environment R^* that may occur between them. For the same reason, the potential intermediate interference has to be inserted between the post-conditions of the two operands to give the post-condition of the entire sequential composition.

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. Because any thread could take the first or last atomic action, the pre-conditions and post-conditions of all threads must tolerate interference from the others. Under these conditions, the final state will satisfy both post-conditions. All pre-conditions and rely conditions must hold, but concurrent combination can guarantee only the disjunction of the separate guarantee conditions.

The corresponding proof rule is in Figure 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 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 .

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

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 $P_{[\text{self} := t]}$ this substitution of *self* by the actual thread identifier t in P . For example the implication $G_1 \Rightarrow R_2$ will become:

$$G_1_{[\text{self} := 1]} \Rightarrow R_2_{[\text{self} := 2]}$$

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)

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

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

$$\begin{aligned} L.\text{lock}() &\sqsupseteq (L.\text{owner} \neq \text{self}, R, G, L.\text{owner} = \text{self}) \\ L.\text{unlock}() &\sqsupseteq (L.\text{owner} = \text{self}, R, G, L.\text{owner} \neq \text{self}) \end{aligned}$$

where

$$\begin{aligned} R &= \text{LockRely} = \text{ID}(L.\text{owner} = \text{self}) \\ G &= \text{LockGuar} = (L.\text{owner} \notin \{\text{self}, \text{null}\} \Rightarrow \text{ID}(L.\text{owner})) \end{aligned}$$

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

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

$$LockGuar_{[self := i]} \Rightarrow LockRely_{[self := j]}$$

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).

$$\begin{aligned} R &= LockRely \wedge (L.\overline{owner} = self \Rightarrow P) \quad \text{and} \\ G &= LockGuar \wedge (L.\overline{owner} \neq self \Rightarrow P) \end{aligned}$$

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 atomically at some instant between the method’s invocation and return. This property is known as *linearisability* [10]. 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 $Preserve(Inv)$).

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

$$\begin{array}{l}
AbsContains(e) : \langle AbsResult := e \in Abs \quad \rangle \\
AbsAdd(e) : \quad \langle AbsResult := e \notin Abs ; \\
\quad \quad \quad Abs := Abs \cup \{e\} \quad \rangle \\
AbsRemove(e) : \langle AbsResult := e \in Abs ; \\
\quad \quad \quad Abs := Abs \setminus \{e\} \quad \rangle
\end{array}$$

Figure 2: Abstract list operations

threads. With our approach, 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 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 $x = \overline{x} + 1$ to the whole *inc* function.

6 Fine-grained list algorithms

In the rest of the paper, we consider a few fine-grain algorithms for which we have applied R-G reasoning. All the algorithms are concurrent, linearisable implementations of the *set* abstract data type presented in Fig. 2. It consists of the operations *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 *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.

<pre> 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 </pre>	<pre> 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 </pre>	<pre> 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 </pre>
--	--	---

Figure 3: Lock-coupling list algorithm

6.1 Pessimistic list

The first algorithm (see Fig. 3) is pessimistic in its concurrency management: it always locks a node before accessing it.³ *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 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.

Even for this relatively simple algorithm, a naïve ownership approach is insufficient. While a node is locked, its owner can modify it arbitrarily as long as it re-establishes the list structure before unlocking. But, additionally, nodes that are reachable from the head, other than the one being modified, must remain reachable (see the last clause in the rely condition hereafter).

In the following representation invariant *ListInv*, the predicate *noOwn*(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 abstraction and re-establishes the invariant when it unlocks

³Our pseudocode uses a Java-like notation, without its more complex features such as object orientation, interrupts, class loaders, etc.

the node.

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

Note that the node allocated in *add* is private until the assignment $n1.\text{next} := n2$; therefore, as explained earlier, there can be no interference in the fields of $n2$ until that point. We write $\text{Node}(n)$ for the assertion that n is a valid public node. Furthermore, we write $n \rightarrow m$ for $\text{Node}(n.\text{next}) \wedge n.\text{next} = m$, 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. 3 and show that (i) the list invariant is preserved by all atomic statements, and (ii) the post-condition of *add* and *remove* is *Result* = *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:

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

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 *locate*(e) (see *add* and *remove* in Fig. 3, *locate* in Fig. 5, and *validate* in Fig. 4). 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).

```

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

```

Figure 4: Optimistic *validate* function

The representation invariant is similar to the one for the pessimistic algorithm, with the exception that the properties hold about all nodes, not only unlocked ones:

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

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).

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

7 Lazy list

In this section, we study a highly concurrent implementation using optimistic and lazy techniques, due to Heller et al. [8], presented in Fig. 5. 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`

<pre> <i>locate</i>(<i>e</i>) : while (true) { pred := Head ; curr := pred.next ; while (curr.val < <i>e</i>) { pred := curr ; curr := curr.next } ; pred.lock() ; curr.lock() ; if <i>validate</i>(pred, curr) then return pred, curr else pred.unlock() ; curr.unlock() } </pre>	<pre> <i>contains</i>(<i>e</i>) : curr := Head ; while (curr.val < <i>e</i>) curr := curr.next ; if curr.marked then return false else return curr.val = <i>e</i> <i>validate</i>(pred, curr) : if ¬pred.marked and ¬curr.marked and pred.next = curr then return true else return false </pre>	<pre> <i>add</i>(<i>e</i>) : same as lock-coupling <i>remove</i>(<i>e</i>) : n1, n2 := <i>locate</i>(<i>e</i>) ; if n2.val = <i>e</i> then n2.marked := true [*C] ; n3 := n2.next ; n1.next := n3 ; Result := true else Result := false [*D] endif ; n1.unlock() ; n2.unlock() ; return Result </pre>
---	---	--

Figure 5: Lazy list algorithm

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, 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.

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 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.

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

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*.⁴ 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.

7.2 Pre-conditions and post-conditions

All the methods share the same pre-condition:

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

The post-conditions of the external methods *contains*, *add* and *remove* are just their abstract implementations given in Fig. 2 (conjoined with *ListInv*), whereas the post-condition of $(\text{pred}, \text{curr}) := \text{locate}(e)$ is given below.

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

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.

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

⁴From the previous line, these nodes are reachable from *Head*.

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 .

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

7.4 Proof of safety

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

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

We now need to prove that the guarantee condition holds for all atomic actions of the algorithm. (1) *LockGuar* holds, because locking is performed using its prescribed interface. (2) $n.next$ and $n.marked$ are updated only for private or locked nodes. (3) Only locked nodes are removed from the list ($n2$ in *delete*). (4) For a public node n , $n.marked$ is never set to false and there are no assignments to $n.val$. Finally, most statements do not update the list, trivially preserving the list invariant *ListInv*. Here are the interesting exceptions:

- In *add*, $n1.next := n2$. The newly created node $n2$ has already been fully initialised, $n1.val < n2.val = e$, and e is also added to *Abs*.
- In *remove*, $n2.marked := \text{false}$. The element e is removed from the abstract set *Abs*.
- In *remove*, $n1.next := n3$. Node $n2$ 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 full paper). 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 all placements 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.

Clearly, the linearisation point lies between the last assignment to $curr$ in the loop and the test of the $.marked$ flag. If $contains$ returns true, then e was a member of the list at the last assignment to $curr$ and when the $.marked$ flag was checked. If, however, $contains$ returns false then we could have two cases:

- e was not in the list when the last pointer in the list was followed. In this case, the linearisation point is the last assignment to $curr$.
- e was in the list ($curr.val = e$), but subsequently marked. In this case, the linearisation point is not necessarily the test of the $.marked$ flag, because another node containing e could have been added to the list in the meantime.

In the second case, however, there always exists a point when e is not in the list, provided the environment cannot atomically mark the node containing e and insert another node containing e in the list. This constraint is captured by conjoining the following predicate to the rely and guarantee conditions:

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

This is a genuine two-state predicate: immediately after the step when the $.marked$ bit is set, n is not in the abstract set. This guarantees that a linearisation point exists; it is exactly after the $.marked$ 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

Much recent research on non-interference in shared-memory concurrent programs has focused on run-time detection [4, 22]. Such tools address a single class of errors and are based on tests and heuristics. They have proved their merits in detecting many errors automatically, but cannot distinguish beneficial interference from race conditions, so they are liable to false alarms when applied to fine-grain algorithms. Furthermore they cannot give a guarantee of correctness.

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.

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 (which appears to be commonly true for practical programs) 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 pre-defined modular structure to a proof that is necessarily global. We argue that the compositionality of R-G has the attraction, from an engineering point of view, to allow reasoning about a single object or thread in isolation from the others. 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 $ID(p)$ and $Preserve(p)$ 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.

The R-G approach is widely used in hardware verification (where it is called assume-guarantee). For instance, Henzinger et al. [9] 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 [7, 8, 18, 19]. However, to our knowledge, no complete formal proof was published before. For instance, the crucial rely condition $Preserve(m \rightarrow^* n)$ was never recognised previously.

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 such fine-grained algorithms is more challenging than reasoning about their coarse-grained predecessors, 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. 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.

Acknowledgments

We thank Leslie Lamport for his insightful comments.

References

- [1] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Prog. Lang. Syst.*, 17(3), May 1995.
- [2] R. Bayer and M. Schkolnick. Concurrency of operations on B-Trees. *Acta Informatica*, 9, 1977.
- [3] J. Dingel. Computer-assisted assume/guarantee reasoning with VeriSoft. In *Proc. Int. Conf. on Software Eng. (ICSE-25)*. IEEE Computer, May 2003.
- [4] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Symp. Principles of Prog. Lang. (POPL)*, 2004.
- [5] C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular verification of multithreaded programs. *Theoretical Computer Science*, 338(1–3), June 2005.
- [6] J. V. Guttag, E. Horowitz, and D. R. Musser. Abstract data types and software validation. *Commun. ACM*, 21(12), 1978.
- [7] T. Harris. A pragmatic implementation of non-blocking linked-lists. *Lecture Notes in Comp. Sc.*, 2180, 2001.
- [8] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. Schierer, and N. Shavit. Lazy lists. Submitted for publication, July 2005.
- [9] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Proc. 10th Int. Conf. on Comp.-Aided Verif. (CAV)*, volume 1427 of *Lecture Notes in Comp. Sc.* Springer-Verlag, 1998.
- [10] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3), July 1990.
- [11] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10), Oct. 1974.
- [12] B. Jacobs, K. R. M. Leino, and W. Schulte. Verification of multithreaded object-oriented programs with invariants. In *W. on Spec. and Verif. of Component-Based Sys.*, Oct. 2004.
- [13] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, 1983.
- [14] L. Lamport. The temporal logic of actions. *ACM Trans. Prog. Lang. Syst.*, 16(3), 1994.
- [15] L. Lamport. Composition: A way to make proofs harder. In *Compositionality: The Significant Difference (Proc. COMPOS'97 Symp.)*, volume 1536 of *Lecture Notes in Comp. Sc.* 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 Dist. Comp.* ACM Press, 1987.

- [18] M. Michael. High-performance dynamic lock-free hash tables and list-based sets. In *Symp. on Para. Algo. and Archi.* ACM Press, 2002.
- [19] M. Michael and M. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared-memory multiprocessors. *J. of Para. and Dist. Comp.*, 51(1), 1998.
- [20] P. W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR*, 2004.
- [21] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5), 1976.
- [22] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4), 1997.



Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
<http://www.inria.fr>
ISSN 0249-6399