

# Reducing Deadlock and Livelock Freedom to Termination\*

Byron Cook<sup>1,2,3</sup>, Stephen Magill<sup>4</sup>, Matthew Parkinson<sup>1</sup>, and Thomas Ströder<sup>5</sup>

- 1 Microsoft Research Cambridge, UK  
`{bycook,mattpark}@microsoft.com`
- 2 Microsoft Research NYC, USA
- 3 University College London, UK
- 4 Unaffiliated  
`stephen.magill@gmail.com`
- 5 RWTH Aachen University, Germany  
`stroeder@informatik.rwth-aachen.de`

---

## Abstract

In this paper we introduce a general method for proving deadlock and livelock freedom of concurrent programs with shared memory. Our goal in this work is to support programs which use locks stored in mutable data structures. The key to our technique is the observation that dependencies between locks can be abstracted using recursion and non-determinism in a sequential logic program such that termination of the abstraction implies deadlock and livelock freedom of the original program.

**1998 ACM Subject Classification** F.3.1 Specifying and Verifying and Reasoning about Programs

**Keywords and phrases** deadlock, livelock, concurrency, termination, shared-memory

## 1 Introduction

Concurrent programs that use locks stored in mutable data structures are especially hard to show deadlock/livelock free, as a static lock acquisition order is difficult to impose given memory allocation, de-allocation and re-allocation. Currently the search for effective automatic methods for this problem remains open.

In this paper we present a procedure that addresses this problem via abstraction to the question of termination of a sequential logic program. We introduce an abstraction method that—using recursion and non-determinism—captures the dependencies between locks in the original concurrent program. Termination of our abstraction implies acyclicity of the lock dependency relation, and hence deadlock freedom. Moreover, termination of our abstraction also implies that threads cannot diverge while holding a lock. Together with the property that locks cannot be acquired infinitely often (which can be proved using existing techniques), this establishes livelock freedom.

To show the viability of our approach, we have successfully applied it to a series of challenging problems, among them a lock-coupling list [10], the optimistic list of Heller et al. [5], the lock-based queue of Michael and Scott [7], and the balanced binary trees of Kung and Lehman [6], Bronson et al. [1], and Schlatter Ellis [8]. The last example is particularly challenging, as it uses different lock types which are inter-dependent. To the best

---

\* This work was supported by Microsoft Research Cambridge.

of our knowledge this represents the first known fully automatable proofs of deadlock/livelock freedom for these published algorithms.<sup>1</sup>

## 2 Approach

Our approach is based on showing two properties: (1) locks cannot be acquired infinitely often and (2) each lock acquired by the program is eventually released again. Together these imply deadlock and livelock freedom of the concurrent program. Note that we assume a fair scheduler, as property (2) is trivially false otherwise.

We use the previous separation logic based techniques underlying the tool CAVE [11] to prove memory safety, data structure consistency, and to learn the possible effects on the shared data structures, expressed as actions [2]. We then use another previous technique [4] to establish that those actions are not executed infinitely often, which already implies (1).

Next we employ our abstraction construction, which is designed to represent the dependencies between the locks as a sequential logic program. The abstraction is based on the fact that whenever a thread  $T_1$  tries to acquire a lock held by another thread  $T_2$ ,  $T_2$  must first reach a corresponding unlock operation before  $T_1$  can continue its execution.<sup>2</sup> If we interpret this dependency of the further execution of  $T_1$  on the execution of  $T_2$  as a control switch from  $T_1$  to  $T_2$  until the latter reaches the matching unlock operation, we can express this dependency as a function call (using lock operations as entry or call points and corresponding unlock operations as exit or return points). However, when switching control from a thread  $T_1$  to another thread  $T_2$ , we do not know which lock operation in  $T_2$ 's code was executed to acquire the lock  $T_1$  is waiting for. Hence, in our abstraction we need to allow a non-deterministic choice that could jump to any lock operation of  $T_2$ . Moreover, we need to pass a representation of the current state of the shared data structure where the lock has been acquired as an argument of each call to establish a relation on the heap locations where locks are acquired. Between the calls to lock operations, our abstraction mimics the behaviour of the program on the shared data structure. Because we allow non-deterministic jumps to arbitrary lock operations, we use the same predicate in the resulting logic program for all lock operations. Thus, we will obtain a recursive and non-deterministic, but sequential program simulating the control switches between the threads whenever they need to wait for a lock to be released. If this program is terminating, we know that no thread has to wait forever to release a lock. Together with an assumption of a fair scheduler, this implies (2): If a thread fails to terminate after acquiring a lock, so will the abstraction. Together with the already established property that lock actions cannot be executed infinitely often, this implies livelock freedom. Furthermore, the lock abstraction forms a disjunctive relation on heap locations representing all possible lock orderings. If the program is terminating, this relation is well-founded and cannot contain circular dependencies. This implies deadlock freedom.

**Example.** Consider the the Java-like program in Figure 1, which implements an add operation in a concurrent finite acyclic list that uses hand-over-hand locking to increase the lock-granularity. The syntax `<c>` is used to indicate the atomic execution of the command `c`.

---

<sup>1</sup> Caveat: Some of the steps in our method depend on previously published tools that are no longer available. Thus, these steps are currently performed manually, but could be automated with additional development work to replace these old tools.

<sup>2</sup> In principle, this is an over-approximation if locks can be released by threads which did not acquire them. However, if  $T_2$  itself releases the lock, we can be sure that it is no longer held by  $T_2$ .

```

1  /* Create a node */ 16 lock(l); 31 unlock(l); 46 unlock(p);
2 Node node(int e, 17 <p = l.head;> 32 unlock(p); 47 unlock(c);
3 Node next) { 18 if (p == null) { 33 return; 48 return;
4 <h = new();> 19 // empty list 34 } 49 }
5 <h.lock = 0;> 20 h = node(e, null); 35 unlock(l); 50 unlock(p);
6 <h.value = e;> 21 <l.head = h;> 36 <c = p.next;> 51 p = c;
7 <h.next = next;> 22 unlock(l); 37 52 <c = p.next;>
8 return h; 23 return; 38 /* hand-over- 53 }
9 } 24 } 39 hand locking */ 54 h = node(e, null);
10 25 lock(p); 40 while (c != null) { 55 <p.next = h;>
11 /* Add e to sorted 26 <v = p.value;> 41 lock(c); 56 unlock(p);
12 list l. */ 27 if (v >= e) { 42 <v = c.value;> 57 return;
13 add(List l, int e) { 28 // new first node 43 if (v >= e) { 58 }
14 Node p,c,h; 29 h = node(e, p); 44 h = node(e, c);
15 int v; 30 <l.head = h;> 45 <p.next = h;>

```

■ **Figure 1** Thread-safe insert into sorted list, managed using hand-over-hand locking.

The type `Node` describes a node in a singly-linked list with fields `lock`, `value`, and `next`. The type `List` is a record with fields `head` and `lock`, the former pointing to the head of the list.

Our aim is to prove deadlock/livelock freedom of a program with an arbitrary finite number of threads, each executing the code in Figure 1 with arbitrary values passed to the `add` operation. The difficulty in proving this concurrent program deadlock- and livelock-free is the hand-over-hand locking seen in the program: it is a common pattern in concurrent programs, and unfortunately it precludes the use of any previously known fully-automatic approach to prove deadlock or livelock freedom.

The actions which can occur in this program are adding (line 21, 30, 45, or 55), locking (line 16, 25, or 41) and unlocking (line 22, 31, 32, 35, 46, 47, 50, or 56) a node in the list. It is easy to see that adding a node cannot be executed infinitely often as the code (and hence each thread) directly terminates thereafter. Lock and unlock actions both occur in the loop starting at line 40, so they occur finitely often if this loop always terminates. The number of loop iterations is bounded by the finite length of the original list plus the finite number of add actions executed by the threads. Hence, no actions can be executed infinitely often, establishing property (1).

We show property (2) automatically by creating a *lock abstraction*, which characterizes the paths from each lock operation to the matching unlock operation. In our example, if we start at a lock operation, we can at most traverse one more lock operation before reaching a matching unlock operation (e.g. from line 41, we either directly reach a matching unlock operation at line 47 or 56 without traversing any further lock operations, or we enter the loop again and definitely reach a matching unlock operation at line 46 or 50 after traversing the lock operation at line 41 again – note that the position in the list where the lock has been acquired originally is referenced by `p` after executing line 51). Moreover, each such lock dependency acquires the second lock at the successor of the node where the first lock has been acquired. Having a finite list which only grows finitely often, this process cannot be repeated infinitely often. Also, as the direction in which further locks are acquired is always the same, it is not possible to form a circular dependency. This shows that eventually every attempt to acquire a lock is successful, which is equivalent to property (2).

This reasoning is captured by the logic program in Figure 2, which results from our construction. Here, we observe that the data structure is represented by terms using the constructors `h/1` and `lh/1` for the head structure, `n/2` and `ln/2` for nodes within the list, and `nil` for the empty list. The respective versions with the prefix `l` denote structure parts which are known to be locked by some thread while the other versions denote an unknown lock

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> lock(D,N). lock(D1,N1) :-   p1617182425(     D1,E1,P1,C1,H1,V1,     L2,E2,P2,C2,H2,V2),   interf(L2,P2,C2,H2,N1,     L3,P3,C3,H3,N2),   lock(P3,N2). lock(D1,N1) :-   p2526273435364041(     L1,E1,D1,C1,H1,V1,     L2,E2,P2,C2,H2,V2),   interf(L2,P2,C2,H2,N1,     L3,P3,C3,H3,N2),   lock(C3,N2). lock(D1,N1) :-   p414243495051524041(     L1,E1,P1,D1,H1,V1,     L2,E2,P2,C2,H2,V2),   interf(L2,P2,C2,H2,N1,     L3,P3,C3,H3,N2),   lock(C3,N2). </pre> | <pre> p1617182425(   h(n(I,R)),E,P,C,H,V,   lh(n(I,R)),E,n(I,R),   C,H,V). p2526273435364041(   lh(n(I1,n(I2,R1))),E,   n(I1,n(I2,R1)),C,H,V,   h(R2),E,ln(I1,n(I2,R1)),   n(I2,R1),H,I1). p414243495051524041(   h(R1),E,   ln(I1,n(I2,n(I3,R2))),   n(I2,n(I3,R2)),H,I1,   h(R1),E,ln(I2,n(I3,R2)),   n(I3,R2),H,I2). interf(L,P,C,H,N,   L,P,C,H,N). interf(L1,P1,C1,H1,N1,   L2,P2,C2,H2,N2) :-   envstep(L1,N1,L3,N3),   interf(L3,P1,C1,H1,N3,     L2,P2,C2,H2,N2). </pre> | <pre> interf(L1,P1,C1,H1,N1,   L2,P2,C2,H2,N2) :-   envstep(P1,N1,P3,N3),   interf(L1,P3,C1,H1,N3,     L2,P2,C2,H2,N2). interf(L1,P1,C1,H1,N1,   L2,P2,C2,H2,N2) :-   envstep(C1,N1,C3,N3),   interf(L1,P1,C3,H1,N3,     L2,P2,C2,H2,N2). interf(L1,P1,C1,H1,N1,   L2,P2,C2,H2,N2) :-   envstep(H1,N1,H3,N3),   interf(L1,P1,C1,H3,N3,     L2,P2,C2,H2,N2). envstep(n(I1,R1),s(N1),   n(I1,R2),N2) :-   envstep(R1,N1,R2,N2). envstep(ln(I1,R1),s(N1),   ln(I1,R2),N2) :-   envstep(R1,N1,R2,N2). </pre> | <pre> envstep(h(R1),s(N1),   h(R2),N2) :-   envstep(R1,N1,R2,N2). envstep(lh(R1),s(N1),   lh(R2),N2) :-   envstep(R1,N1,R2,N2). envstep(R1,s(N),R2,N) :-   addH(R1,R2). envstep(R1,s(N),R2,N) :-   addHN(R1,R2). envstep(R1,s(N),R2,N) :-   addNN(R1,R2). envstep(R1,s(N),R2,N) :-   addN(R1,R2). addH(h(nil),   h(n(I2,nil))). addHN(h(n(I1,R)),   h(n(I2,n(I1,R)))). addNN(n(I1,n(I2,R)),   n(I1,n(I3,n(I2,R)))). addN(n(I1,nil),   n(I1,n(I3,nil))). </pre> |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

■ **Figure 2** Lock abstraction whose termination implies deadlock/livelock freedom.

status. Using this term representation, the `lock` predicate encodes the lock dependencies. Its arguments are the state of the data structure where a lock is acquired and a prophecy counter stating how many operations can still be performed by the interfering environment (*i.e.* other threads). The use of the latter is motivated by the fact that all actions are known to be executed only finitely often. Next, a postcondition transformer (starting with `p` and displaying the two digits line numbers of the path it follows in its name) is used to mimic the program behaviour along a path on the program variables. Afterwards, the `interf` predicate is used to apply the four add actions (corresponding to the four locations where nodes are added) non-deterministically to the data structure representations pointed to by the program variables (variables which provably never point to the shared data structure are excluded) while the prophecy counter is reduced accordingly. Finally, we have the recursive call to the `lock` predicate. Its first argument is the respective program variable passed to the lock operation which might block the current thread's execution. Its second argument is the number of remaining interference operations. The dependency described above is encoded by the last clause for the `lock` predicate.

The lock abstraction captures all execution paths that start at a lock operation and traverse further lock operations before reaching the matching unlock operation for the initial lock operation. Termination of the lock abstraction program thus implies that each acquired lock must eventually be released. This implies deadlock freedom and, together with the fact that locks cannot be acquired infinitely often, also implies livelock freedom.

To prove termination automatically, we apply the termination prover AProVE [9, 3], which proves the abstraction in our example terminating in 15 seconds. Our method scales to bigger programs, as the complexity of the abstraction is dependent on the number of actions and the size of the critical sections protected by the locks (*i.e.* those parts of the program between lock and corresponding unlock operations). Both are typically small in practice to allow as much concurrency as possible.

### 3 Conclusion

The problem of proving deadlock and livelock freedom is made difficult by the level of cross thread lock dependencies. In this paper we have presented an abstraction technique that encodes the lock dependencies from the concurrent program into a sequential logic program

such that existing termination tools prove livelock and deadlock freedom. Our approach has been successfully applied to a series of challenging problems, including a lock-coupling list [10], the optimistic list of Heller et al. [5], the lock-based queue of Michael and Scott [7], and the balanced binary trees of Kung and Lehman [6], Bronson et al. [1], and Schlatter Ellis [8].

**Acknowledgements** We would like to thank the following people for making valuable comments on a draft of an extended version of this paper: Aws Albarghouthi, Marc Brockschmidt, James Brotherston, Isil Dillig, Thomas Dillig, Carsten Fuhs, Jürgen Giesl, Kareem Khazem, Heidy Khlaaf, Zachary Kincaid, Eric Koskinen, Rustan Leino, Peter O’Hearn, and Viktor Vafeiadis. We would also like to thank Viktor Vafeiadis for his support on using CAVE.

---

## References

---

- 1 Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP, pages 257–268, New York, NY, USA, 2010. ACM.
- 2 Cristiano Calcagno, Matthew Parkinson, and Viktor Vafeiadis. Modular safety checking for fine-grained concurrency. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis*, volume 4634 of *Lecture Notes in Computer Science*, pages 233–248. Springer Berlin Heidelberg, 2007.
- 3 Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes, and Carsten Fuhs. Symbolic evaluation graphs and term rewriting: a general methodology for analyzing logic programs. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*, PPDP ’12, pages 1–12, New York, NY, USA, 2012. ACM.
- 4 Alexey Gotsman, Byron Cook, Matthew Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don’t block. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL ’09, pages 16–28, New York, NY, USA, 2009. ACM.
- 5 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. In James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer, editors, *Principles of Distributed Systems*, volume 3974 of *Lecture Notes in Computer Science*, pages 3–16. Springer Berlin Heidelberg, 2006.
- 6 H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems*, 5(3):354–382, September 1980.
- 7 Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC ’96, pages 267–275, New York, NY, USA, 1996. ACM.
- 8 Carla Schlatter Ellis. Concurrent search and insertion in AVL trees. *IEEE Transactions on Computers*, C-29(9):811–817, 1980.
- 9 Peter Schneider-Kamp, Jürgen Giesl, Alexander Serebrenik, and René Thiemann. Automated termination proofs for logic programs by term rewriting. *ACM Transactions on Computational Logic*, 11(1):2:1–2:52, November 2009.
- 10 Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge Computer Laboratory, 2008.
- 11 Viktor Vafeiadis. RGSep action inference. In Gilles Barthe and Manuel Hermenegildo, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 5944 of *Lecture Notes in Computer Science*, pages 345–361. Springer Berlin Heidelberg, 2010.