Towards automatic proofs of lock-free algorithms

Loïc Fejoz* Stephan Merz
INRIA Nancy & LORIA, Nancy, France
Loic.Fejoz@loria.fr Stephan.Merz@loria.fr

Abstract

The verification of lock-free data structures has traditionally been considered as difficult. We propose a formal model for describing such algorithms. The verification conditions generated from this model can often be handled by automatic theorem provers.

1 Lock-free data structures and their verification

Modern processors make more and more use of parallel processing to gain computing power, be it via multi-core architectures or by the use of specialized co-processors for graphics or signal processing. It is well known that access to shared memory by several threads requires protection so that operations of different threads do not interfere. Traditionally, mutual-exclusion locks have been used to protect shared data structures in the presence of multi-threading. However, the extensive use of locks is not without problems: coarse-grained locks tend to create points of contention and degrade performance, whereas fine-grained locks are prone to deadlocks and priority inversion.

Lock-free data structures [2, 8, 9] present an interesting alternative approach to concurrency control. They are based on operations guaranteed to be executed atomically by the processor. Compare-And-Swap (CAS), Load-Link/Store-Conditional (LL/SC), and Test-And-Set (TAS) are examples of such operations. It has been shown [5] that their use can make algorithms scale better, based on an optimistic approach to concurrency control. Lock-free algorithms are also the basis for implementing higher-level abstractions for concurrency, such as Software Transactional Memory (STM, [7]).

Because of their importance and intricacy, we believe that algorithms for lock-free data structures are a prime application area for formal methods. Currently, the correctness of such algorithms is usually justified informally, sometimes complemented by model checking over small instances. The correctness criterion most frequently required for lock-free data structures is *linearizability*: the externally observable behaviour should correspond to some linearization of atomically executed operations. In this way, a data structure can be described sequentially, using atomic operations; all invariants proved about this atomic specification will be preserved in the implementation. Formally, proving linearizability amounts to proving a refinement relationship between the concurrent implementation and the sequential specification.

While existing formalisms and tools can be used to describe the algorithms and to prove refinement, we have found that this tends to result in a messy encoding and/or generates long formulas that can be difficult to prove. We therefore propose a custom framework for proving the correctness of such algorithms, preferably using automatic proof tools.

*This work was supported by Microsoft Research through its European PhD Scholarship Programme.

```
int addN(int *x, int n) {
  Descriptor d;
  d->x = x;
  d->n = n;
  do {
    d->o = *(d->x);
    // (try to) install the descriptor:
    r = CAS1(d->x, d->o, d);
    if (isDescriptor(r)) {
      // remove existing descriptor
C1:     CAS1(r->x, r, r->o + r->n);
    }
  } while (isDescriptor(r));
  // descriptor has been installed
  // perform operation and remove descriptor
C2: CAS1(d->x, d, d->o + d->n);
  return d->o;
}

int readN(int *x) {
  do {
    r = *x;
    if (isDescriptor(r)) {
C3:     CAS1(r->x, r, r->o + r->n);
    }
  } while (isDescriptor(r));
  return r;
}
```

Figure 1: Implementation of addN with descriptors.

2 Modeling lock-free algorithms

Lock-free algorithms are usually based on an optimistic approach to concurrency control: processes behave as if there were no interference and then check if the operation succeeded. The low-level primitives they rely on are atomic operations such as compare-and-swap (CAS). Single-word CAS is available on many processor architectures and has been shown to be universal [4] in the sense that it can be used to implement similar atomic operations, including multi-word CAS [3]. Its effect is to execute the following code atomically:

```
word_t CAS1(word_t *a, word_t o, word_t n) {
  old = *a;
  if (old == o) { *a = n; }
  return old;
}
```

Descriptors are often helpful for implementing optimistic concurrent algorithms. Descriptors are data structures in which the initiating thread stores all the information necessary to perform the operation. The thread first installs the descriptor, using a CAS operation. When a concurrent thread attempting an operation concerning the same memory cell finds a descriptor instead of an ordinary value, it can either back off, waiting for the first thread to finish, or even complete the operation on its own. As a toy example, consider the operation *addN* that increments a shared variable x by a value n, together with a trivial read operation. Its sequential behavior is described by the following pseudo-code:

```
int addN(int *x, int n) {
  r = *x; *x += n;
  return r;
}
```

```
int readN(int *x) {
  return *x;
}
```

A descriptor-based non-atomic implementation of these operations is presented in Fig. 1. Observe how the implementations of both operations take into account the presence of descriptors, performing the required operation on behalf of the process that installed the descriptor.

In general, when designing a lock-free implementation of a data structure, all of its basic operations have to be considered simultaneously. For example, in the case of a linked-list structure, operations such as *append*, *remove*, and *isEmpty* should be specified. Any concurrent execution of these operations has to be linearizable, and their possible interferences have to be taken into account.

In our model, specifications describe several *algorithms*. A system consists of an arbitrary (finite) number of processes that execute concurrently; each process executes some algorithm. The system's

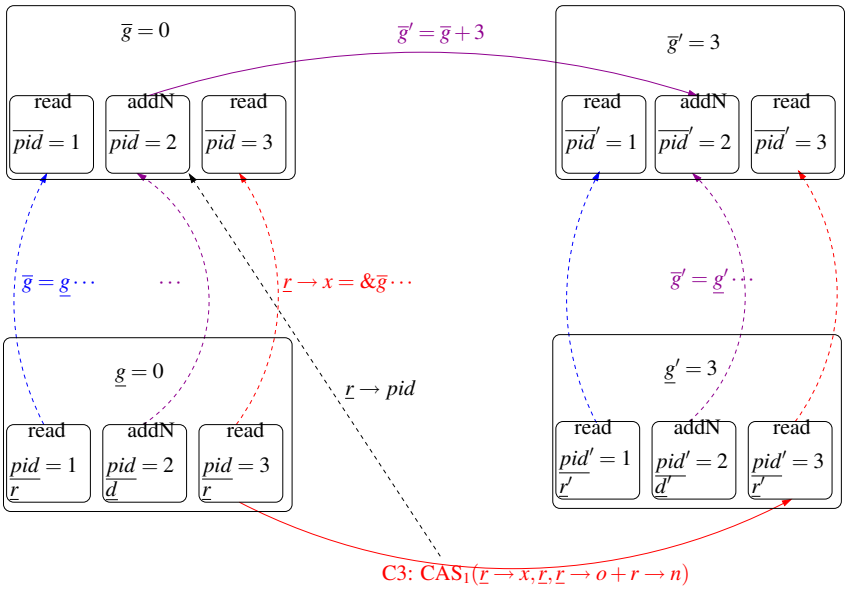

Figure 2: Main refinement verification condition for a 3-process system.

overall state space is given by a set of global variables and local variables for each process. A *partial state* consists of the values of the global variables and the values of the local variables of a process. Each algorithm is specified as a transition system; more precisely, an algorithm specification provides:

- a set of places (control points) of the algorithm, including an initial place,
- an initial condition constraining the initial partial states of any process executing this algorithm,
- a transition relation relating places and partial states before and after the transition,
- a local invariant associating a predicate over partial states with each place, and
- a rely condition for each place that describes the allowed operations of other processes.

The main idea is to provide localized descriptions of algorithms so that refinement can be established step by step. The invariants associated with places help us in these proofs because we can use them when proving refinement. The rely conditions are similar to the assume part of assume-guarantee tuples [6]; they limit the possible interference between processes and thus help composing algorithms.

3 Formal proofs

There are two levels at which proofs have to be performed. First, we ensure that a specification is well-formed by proving a number of standard verification conditions. In particular, the initial condition must establish the invariant of the initial place, and local invariants must be preserved by transitions and by the rely conditions.

More interesting for our purposes is to compare two specifications that describe the same set of algorithms, but at different levels of detail. In particular, the abstract specification typically describes the atomic execution of each algorithm (using two places), and the concrete one describes a lock-free implementation. The proof obligations we define ensure that the concrete specification describes a linearizable refinement of the abstract one.

To describe the refinement relation, we map concrete places to abstract places and define a gluing invariant that constrains the abstract and concrete partial states at each place. Also, for each concrete

transition we indicate the abstract process to which this transition corresponds. This correspondence implicitly defines the linearization points of the implementation. It need not be defined statically, but may also depend on the concrete-level partial states before and after the transition. For example, in the case of a descriptor-based implementation, the descriptor could indicate the process on behalf of which the operation is performed. This idea is illustrated in Fig. 2.

We have formalized this specification framework in the proof assistant Isabelle/HOL and have proved that our refinement conditions ensure linearizability. This formalization helped us to better understand some subtleties of the correctness conditions.

We have also implemented a verification condition generator and have experimented with automatic provers on several examples. Because verification conditions are local to individual places or transitions, they tend to be relatively small formulas. Their complexity and amenability to automatic proof of course depends on the theories that underlie the description of the algorithms. Our initial experiments concern algorithms similar to those of Harris et al. [3] that compare memory references and copy values. In these cases, all proof obligations could be discharged using standard first-order theorem provers such as Spass [10] or SMT solvers such as haRVey [1]. We intend to confirm this encouraging experience by studying realistic implementations of lock-free data structures.

References

[1] D. Déharbe, P. Fontaine, S. Ranise, and C. Ringeissen. Decision procedures for the formal analysis of software. In K. Barkaoui, A. Cavalcanti, and A. Cerone, editors, *Intl. Coll. Theoretical Aspects of Computing (ICTAC 2007)*, volume 4281 of *Lecture Notes in Computer Science*, pages 366–370, Tunis, Tunisia, 2007. Springer. See also `http://harvey.loria.fr/`.

[2] H. Gao. *Design and verification of lock-free parallel algorithms*. PhD thesis, University of Groningen, Apr. 2005.

[3] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Symposium on Distributed Computing*, 2002.

[4] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

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

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

[7] N. Shavit and D. Touitou. Software transactional memory. In *Distributed Computing*, pages 204–213, 1995.

[8] H. Sundell and P. Tsigas. Lock-free and practical deques using single-word compare-and-swap. Technical Report 2004-02, Computing Science, Chalmers University of Technology, Mar. 2004.

[9] J. D. Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, pages 64–69, Las Vegas, NV, 1994.

[10] C. Weidenbach, R. A. Schmidt, T. Hillenbrand, R. Rusev, and D. Topic. System description: Spass version 3.0. In F. Pfenning, editor, *21st Intl. Conf. Automated Deduction (CADE 2007)*, volume 4603 of *Lecture Notes in Computer Science*, pages 514–520, Bremen, Germany, 2007. Springer.