Starling: Stress-Free Concurrency Verification

Matt Windsor[1], Mike Dodds[1], Ben Simner[1], and Matthew J. Parkinson[2]

[1] University of York
{mbw500, mike.dodds, bs829}@york.ac.uk
[2] Microsoft Research Cambridge
mattpark@microsoft.com

Abstract. We introduce *Starling*, a new tool for proving properties of concurrent programs. Starling is based on *program logics*, a form of mathematics for verifying programs in which proofs are close to the program code being verified. This, coupled with its focus on automating as much proof work as possible and a minimalist approach to its construction, make Starling a useful step towards stress-free concurrency verification.

Keywords: Concurrency, verification, automation, program logics.

1 Introduction

Programmers working on operating systems, servers, and other performance-critical scenarios need to tap into the full power of modern CPUs. These CPUs contain multiple *cores*, each capable of simultaneously executing a separate piece of code. Writing programs to work with more than one core is thus a key skill.

The technique of writing programs such that they are able to run on multiple cores at the same time is known as *concurrency*. This is often achieved by dividing the program into multiple separate parts, called *threads*, that run concurrently—perhaps on different cores—and communicate by sharing memory.

Locks and mutual exclusion. Concurrency introduces issues that do not exist in normal programming. Consider, for example, a program with two threads A and B: both are trying to increment a variable x 10 times using the code

```
i = 0; while (i < 10) { x = x + 1; i = i + 1; }
```

If x is 0 initially, one would expect the result to be 20. Yet, any value from 0 to 20 could be left in x after both threads finish. Why? The command `x = x + 1` does three things: it reads x, increments it, and stores back to x. Suppose A reads x and finds it to be 15, but B then performs its increase of x by 1. When A continues, it still thinks x is 15, though it is now 16. It will write $15 + 1$, ie 16, to x, losing B's contribution to x. A has thus *interfered* with B.

This is a common problem in concurrency. One solution is *mutual exclusion*: when one thread has access to x, it forbids others from also having it. We can achieve this using *locks*: algorithms that enforce mutual exclusion by making threads acquire permission to proceed with variable- accessing code before doing so, and relinquish said permission when they no longer need the variable.

As an example of such an algorithm, we consider the *ticket lock*. It works by analogy to a ticket dispenser at, say, a delicatessen:

1. The lock holds a series of tickets labelled $0, 1, \ldots, \infty$. At any stage, it has dispensed n of these tickets: the next one will be labelled n. Of these, k have already taken the lock: the ticket being 'served' is labelled k.
2. A thread wishing to take the lock must first take the ticket, causing n to increase by one. Suppose the ticket it chose was t.
3. The thread must now wait for the lock to serve its ticket. It does so by constantly checking the current value of k, making a note of it (say, as the variable s), and checking whether $s = t$.
4. When $s = t$, the thread acquires the lock.
5. To release the lock, the thread forfeits the ticket, causing k to increase by one: the next ticket may now take the lock.

Proving mutual exclusion for locks. Locks reduce the burden of proving that code is free of unwanted interference, but do not eliminate it. We must prove that the lock satisfies the mutual exclusion property: for locks, this simply means we cannot ever observe the lock being held twice. To do so, we must state said property formally. A first approximation in predicate logic might be

$$holdLock \wedge holdLock \implies false$$

but we have lost the fact that the two *holdLock*s are distinct here.

Finding a valid encoding of mutual exclusion is only the first step towards proving a lock satisfies it. We must show not only that the lock correctly performs the right actions to behave like a lock, but also that the interference of (well-behaved) third parties cannot break the lock. We want a proof technique that

- allows us to specify mutual exclusion in a way that reflects our intuition;
- allows us to reason about interference in a way as simply as possible;
- is easy to use, close to the original code, and automated where possible.

One route forwards is to use a *concurrent program logic*. Based on Hoare's logic for proving properties of non-concurrent programs [3], these combine a formal system for describing shared-memory assertions on shared state with a logical basis for reasoning about whether the actions of a program fulfil such assertions. This line of research has been so fertile that there has been concern [5] over the redundancy in the workload of proving the resulting logics sound. This has led to research on creating logical frameworks, such as the *Concurrent Views Framework* (CVF) [1], to captures the similarities between logics.

We believe that many of the criteria above can be met with relatively little work by turning this process on its head. Instead of using CVF as a base for proving soundness of a large, complex logic, we instead create a small, simple program logic—designed for easy automation—through minimal additions to CVF proper. The rest of this extended abstract discusses the resulting tool.

2 Starling: a tool for proving programs

Starling is a proof tool for a minimal CVF-based program logic. Starling is designed to be fully automatic, to hide (or avoid) as much mathematical complexity as possible, and to be a viable formal proof tool despite this.

Starling is based on the *axiom soundness* rule from CVF: informally, proofs are a series of checks over each command in an program. We must show that each command not only behaves according to assertions made before and after it, but also cannot break any valid assertions the rest of the program may make.

Proving the ticket lock in Starling. We demonstrate Starling with a proof of mutual exclusion for the ticket lock. Starling proofs begin with an *outline* of the program to prove. These have a syntax similar to that of the *C* language, but with some additions. First, the `<command>` notation marks `command` as an *atomic action*: observers cannot see any state between its commencement and its completion. (All actions affecting shared state must be atomic.)

Second, Starling expects a *view*—an assertion of the form {| *A(x) * B(y) * ⋯ * Z(n)* |}—between each command in the outline. Each component of a view, called an *atom*, represents some fact about the shared state that must hold at that point. The *** operator conjoins atoms into views (we discuss how below). Views can contain no atoms (*emp*), or be conditional on some local observation (*if s == t then holdLock() else holdTick(t)*).

We can now write a Starling outline of the ticket lock:

```
shared int n, k; thread int t, s;

method lock() {
  {| emp |}
    <t = n++>; // Fetch and increment ticket in one action.
  {| holdTick(t) |} // By doing the above, we now hold ticket t.
    do {
      {| holdTick(t) |}
        <s = k>;  // Repeatedly fetch and check k against t.
      {| if s == t then holdLock() else holdTick(t) |}
    } while (s != t);
  {| holdLock() |}
}

method unlock() {
  {| holdLock() |}
    <k++>; // Increment
  {| emp |} // We no longer claim we hold a lock.
}
```

We now need to define what these views mean. First, we define the atoms in use:

```
view holdTick(int t);  // we know we hold a ticket with number 't'
view holdLock();       // we know we hold a lock
```

This tells Starling that the alphabet of atoms is *holdTick(t)* where `t` is an integer, and *holdLock()*. Then, we can give views meaning by assigning them to Boolean *constraints*, which filter the possible shared states in which that view can be observed. We can thus realise our earlier intuition about mutual exclusion:

```
constraint holdLock() * holdLock() -> false;  // Mutual exclusion!
```

Starling's power comes from the fact that we can assign any view, or part thereof, a meaning greater than the sum of its parts. This extends to the empty view. We assign a predicate to *emp* to make it invariant (it must *always* hold), then finish by adding more constraints needed for the proof:

```
constraint emp -> n >= k;  // This always holds.
constraint holdTick(a) -> n > a;
constraint holdLock() -> n != k;
constraint holdLock() * holdTick(a) -> k != a;
constraint holdTick(a) * holdTick(b) -> a != b;
```

This concludes our proof of mutual exclusion of the ticket lock, which Starling can now check automatically. This is done through Starling's axiom soundness proof rule, which can be shown to be a strengthening of the CVF rule informally stated earlier. For each command and control-flow transition C in a program, we find the view P immediately before it, and the view Q immediately after. Then, for each `constraint` R `->` $D(R)$, we prove that

$$[\![C]\!] \wedge \lfloor P * (Q \setminus R) \rfloor \implies D(R)$$

where $[\![C]\!]$ takes the semantics of C as a two-state predicate, $\lfloor \_ \rfloor$ maps views to their predicate meanings by collecting the right-hand sides of the applicable `constraint`s, and $\setminus$ subtracts one view from another (ie. an adjoint of $*$).

When all `constraint`s are known, each term above is a simple Boolean predicate and can be checked quickly by *satisfiability-modulo-theories* solvers such as *Z3* [4]. The above proof rule also resembles a Horn clause, with the views as uninterpreted functions. To check a system of such clauses, we need not define the views: a Horn solver such as *HSF* [2] will infer valid definitions if it can. In Starling, we exploit this by allowing `constraint` bodies to be replaced with `?`. This asks Starling to infer those constraints, then try to prove axiom soundness with them. This relieves the proof writer of a large amount of the proof burden.

3 Conclusion

We have demonstrated Starling using the ticket lock. We have also used Starling to prove properties of spinlocks, Peterson's algorithm, and a lock-free version of the multiple-update problem discussed earlier. These additional examples, as well as Starling itself, are available at `https://github.com/septract/starling-tool`.

Future work includes adding support for variable types beyond integers and Booleans, making Starling's frontend more accessible to programmers, and supporting more proof backends. We would also like to formalise the additions and changes we have made to the Views framework. However, as a proof of concept, we feel that Starling is already successful.

References

1. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M., Yang, H.: Views: Compositional reasoning for concurrent programs. POPL (2013)
2. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 405–416. PLDI '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2254064.2254112
3. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (Oct 1969)
4. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver, pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-78800-3_24
5. Parkinson, M.: The next 700 separation logics. In: Leavens, G., O'Hearn, P., Rajamani, S. (eds.) Verified Software: Theories, Tools, Experiments, Lecture Notes in Computer Science, vol. 6217, pp. 169–182. Springer Berlin Heidelberg (2010)