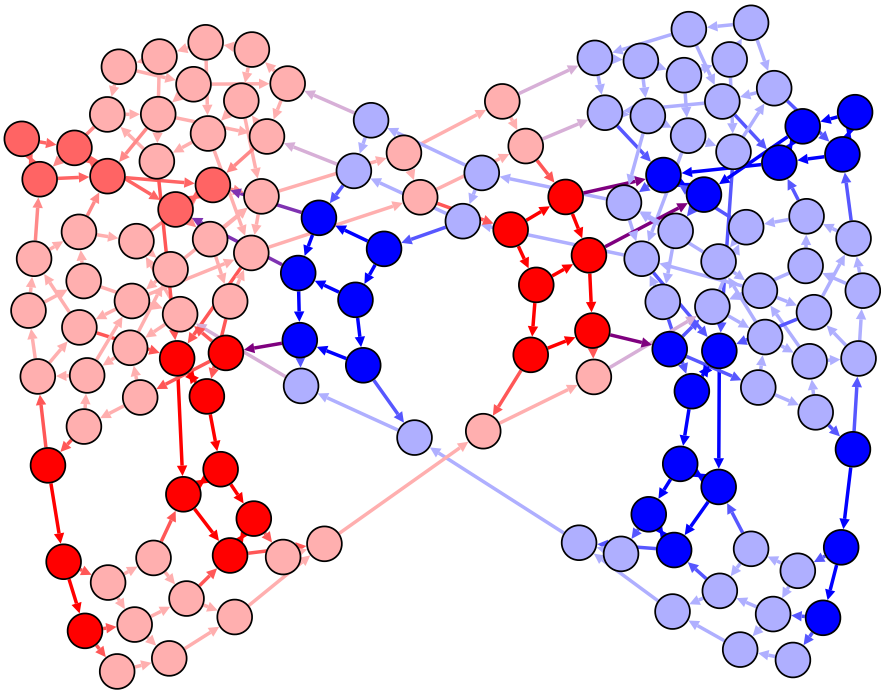


Dekker's Algorithm

2020, MAR 08

Concurrency Modelling

Overview

This article covers Dekker’s algorithm, one of the first correct thread synchronization methods. It is still relevent today for its applicability on general hardware, and more recently, due to it’s use of fundamental ideas that are needed for multithreading and lock-free program design.

In trading systems we have high-speed event based systems requiring multi-threaded programs with close cooperation between threads. This is the bread and butter of high-frequency trading systems which must balance high-speed algorithms for latency and batch processing. Underneath this lies proof of correctness, fast vs slow waiting mechanisms, and real-world Markov chain performance.

There is currently a big push into cyber security and program bugs. This leads to the need for formal methods for program correctness. In this article we consider a simple concurrency (multi-threading) problem, with a solution that is quick to state but exceptionally difficult to prove. The history of concurrent algorithm design is littered with algorithm failures and invalid proofs. There are even current textbooks which have invalid proofs!

Dekker’s Algorithm

Let’s start simple. We’ll analyse a simple cooperation mechanism between two threads, called Dekker’s algorithm. The description is short and uncomplicated. Surely this will be a straightforward exercise to analyse and prove correct? And its finite state machine representation will be compact?

Dekker's Algorithm is a mutual exclusion algorithm for two processes or threads. It ensures that only one process is able to enter and execute the given critical section (code we choose due to the use of shared variables). Dekker's algorithm is a user-space algorithm (these are more efficient and don't require the OS kernel to arbitrate).

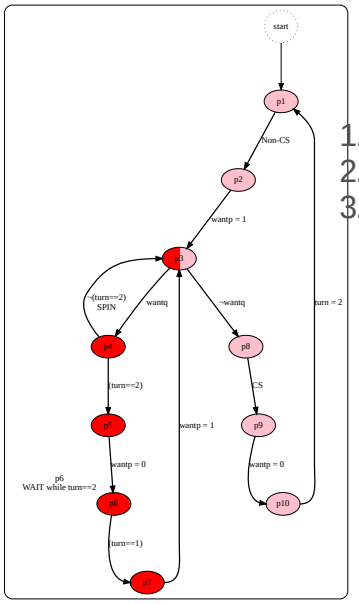
Dekker's algorithm may be stated very simply as

> A process P can enter the critical section if the other does not want to enter,
> otherwise it may enter only if it is its turn.

OK, I'm paraphrasing it here. Lurking within this are the choices of wait or polling loops for the process whose turn it is, and for the other. For a single CPU system or one under load, we can have surprisingly different behaviours, near starvation in some cases. The intricate part of Dekker's algorithm solves the problem of contention fairly. Where both indicate their wish to enter the critical section, the process whose turn it is will busy-wait *spin* until there is no contention, whilst the other process will yield by withdrawing its wish to enter and waiting/yielding (technically it could busy-wait by having its polling spin loop). To do this the method relies on what is called a *strongly fair* scheduler.

Now the program graph (a finite state machine with state variables) is small and straightforward:

Dekker's Algorithm	
`bool wantp = 0, wantq = 0` `int turn = 1`	
thread p	**thread q**
`loop:`	`loop:`
`p1: non-critical section`	`q1: non-critical section`
`p2: wantp = 1`	`q2: wantq = 1`
`p3: while wantq`	`q3: while wantp`
`p4: if turn==2:`	`q4: if turn==1:`
`p5: wantp = 0`	`q5: wantq = 0`
`p6: await turn==1`	`q6: await turn==2`
`p7: wantp = 1`	`q7: wantp = 0`
`p8: critical section`	`q8: critical section`
`p9: turn = 2`	`q9: turn = 1`



This all looks very straightforward, we see from the program graph that it has three loops:

1. Progress through the critical, in pink
2. Spin busy-wait, in red
3. Sleep blocking-wait, in red

Simple right? Ten states for each of the threads? We'll find out when we consider the actual state machine for the system.

Fine print

Given that the default Dekker algorithm uses a spin lock, and this under one scenario competes with the critical section, the algorithm then relies on the critical section being of very short duration.

Other synchronization algorithms skip tracking whose turn is next, which breaks the starvation property, albeit, this depends more heavily on scheduler and system properties in practice.

Analysis and Factorization

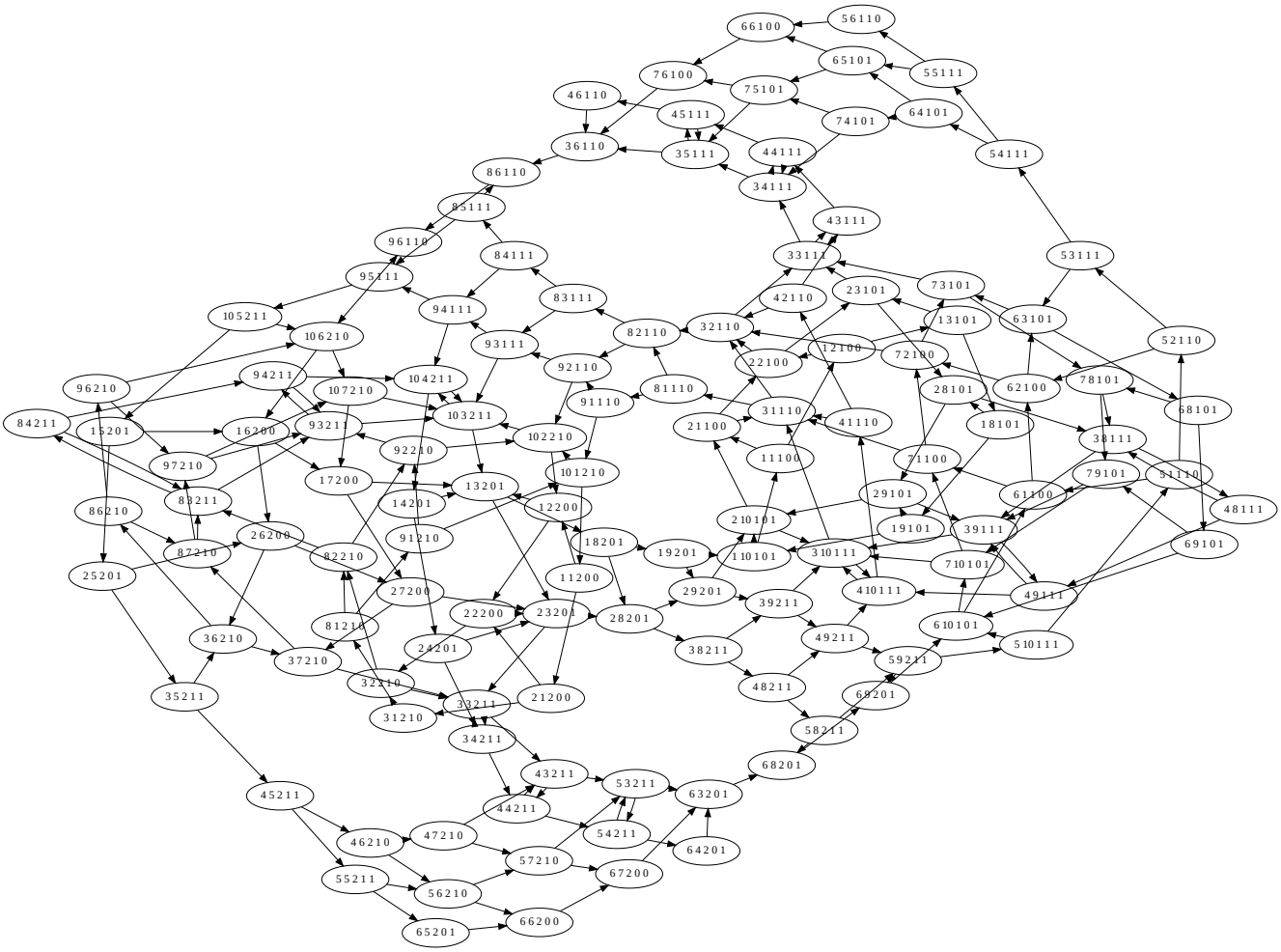
We'll be analysing and writing finite state machine transition often, so let's introduce a shorthand for the state, a 5-tuple in this case:

```
p10: wantp = 0 | q10: wantq = 0

(p-loc, q-loc, turn, wantp, wantq)
```

where p-loc is the program location counter p1 to p10, and similarly for q-loc. For example, we start at (1, 1, 1, 0, 0) which means statement p1, statement q1, turn=1, wantp=0 (false), wantq=0 (false).

We've seen that the high-level description of the algorithm is concise and the program graph is concise, as shown above. However, when we consider visually the full transition systems generated by these two program graphs then the actual picture becomes substantially more complex. The graph below shows this:



The graph above is the finite state machine of the system, formally called the **transition system**. It completely describes the program. Given two exiting transitions, the only choices made are by the scheduler whenever a state has more than one transition. In fact what is drawn above has been simplified, as we'll see next. I'll justify this approach in the next section on the *abstraction model of program interleaving*.

This graph shows the complexity of the system and explains why historically concurrency algorithm designers made mistakes along the way (creating graphs like this if you were particularly unlucky). Our program graph's 10 'states' actually ballooned to 154 program states!

The Abstraction Model of Arbitrary Interleaving

The above graph shows the simplified transition system for only two threads and already it is very complicated. Imagine if we were looking at a program with *ten threads*: each state in general would have 45 exit transitions!

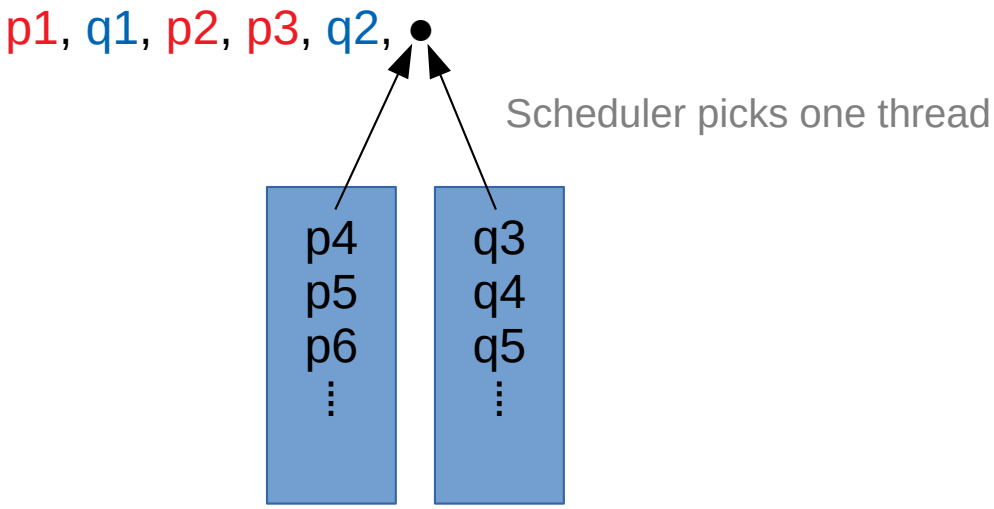
For the purpose of simplifying the analysis I will introduce the definition of the **arbitrary interleaving**model. The key reason here for doing this is that it allows us to greatly simplify the analysis whilst still keeping all states reachable in our original transition system (FSM). The arbitrary interleaving model should be explained in its own right as there is much more interesting structure to discuss. However, I will leave this for another time.

Given a set number of threads, we define the abstraction model as follows:

```
the scheduler chooses one of the available threads and execu
tes its atomic statement;
no other thread will execute until the atomic statement comp
letes.
```

What this means is that the scheduler *picks* some arbitrary (random) interleaving and creates a single threaded program. The key point to remember here is that our job as algorithm designers is to ensure *ALL* of these alternatives are correct!

For example, a two thread program performs the following each time an atomic statement is run:



The transition system generated by the arbitrary interleaving model leaves only the exogenous choice of path to the scheduler. It is the algorithm writers duty to ensure the correctness and the desired properties for all executions, and as we'll see, the correct behaviour for each subgraph.

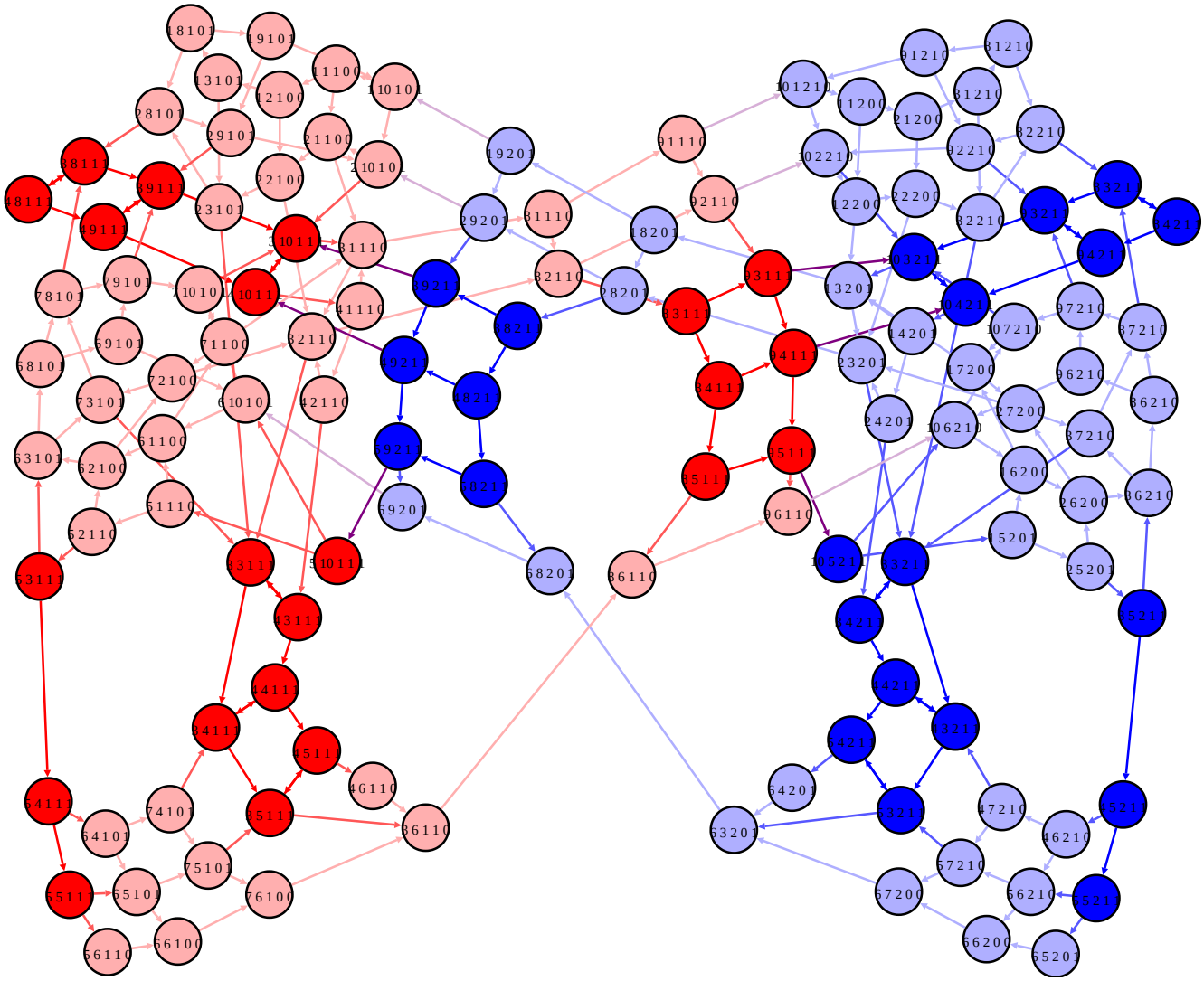
What is remarkable about the transition system generated by the arbitrary interleaving model is that it has eliminated concurrency! Previously with the program graph, we would have had to consider at each point the actions for each of the other threads, and this is exceptionally hard to reason about. The historic literature on mutual exclusion is littered with failures. We now have a sequential program, where we can represent the choice of path by input symbols being simply which process the scheduler picks p or q; or more generally p1, p2, …, pn. I'll show how we can break this fairly large problem into a more modular one using subgraphs.

To reiterate, whilst an actual program path/trace is decided by the external scheduler, we must check all paths of the transition system graph.

Model of Dekker’s Algorithm

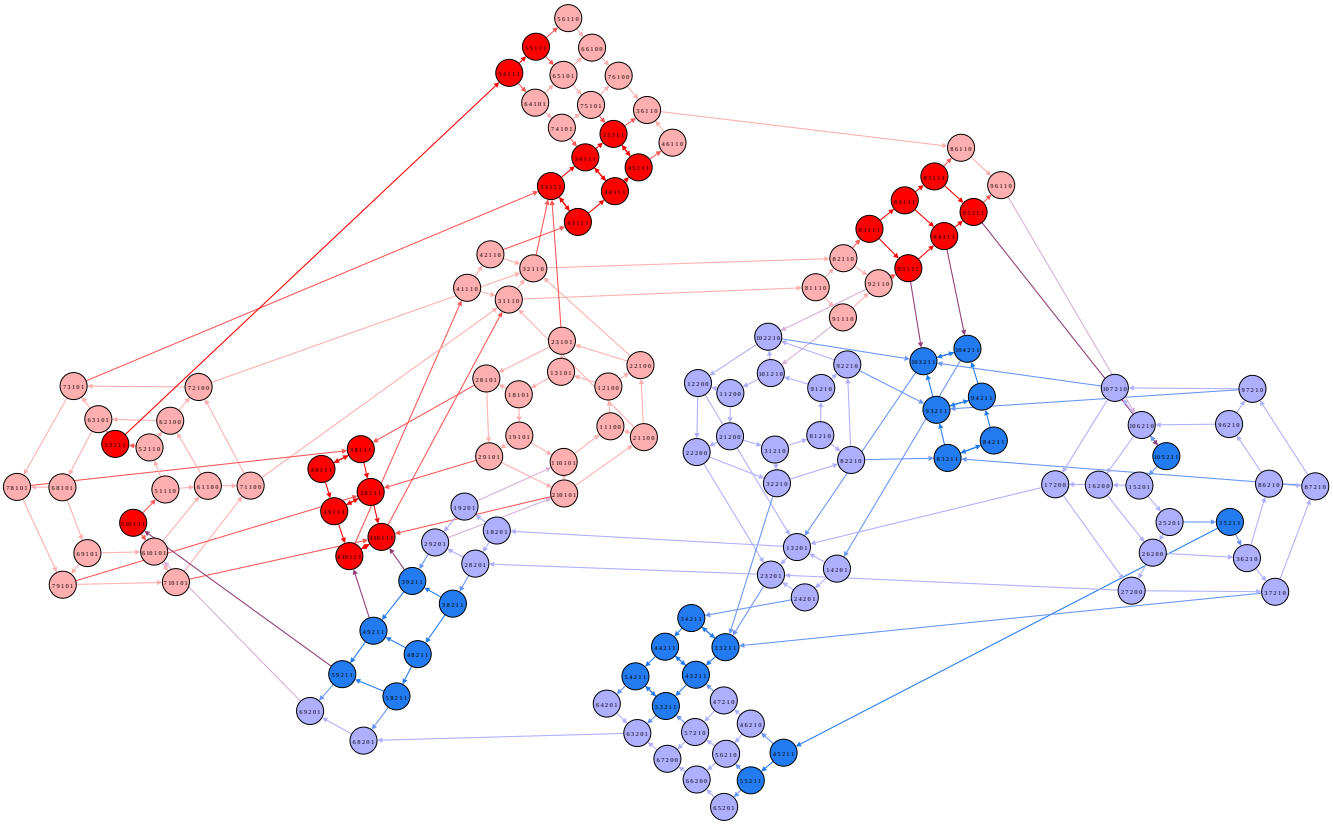
Our aim with Dekker’s algorithm is to simplify the transition system in a number of ways to show the structure of the algorithm and how it differs to its program graphs. With this we will discover all spin loops, waits and states dependent on the scheduler.

Firstly, we can consider colouring by variable `turn` and highlighting contention regions. Here we have red for turn=1 and blue for turn=2. Spin and wait loops are highlighed in bold colours.

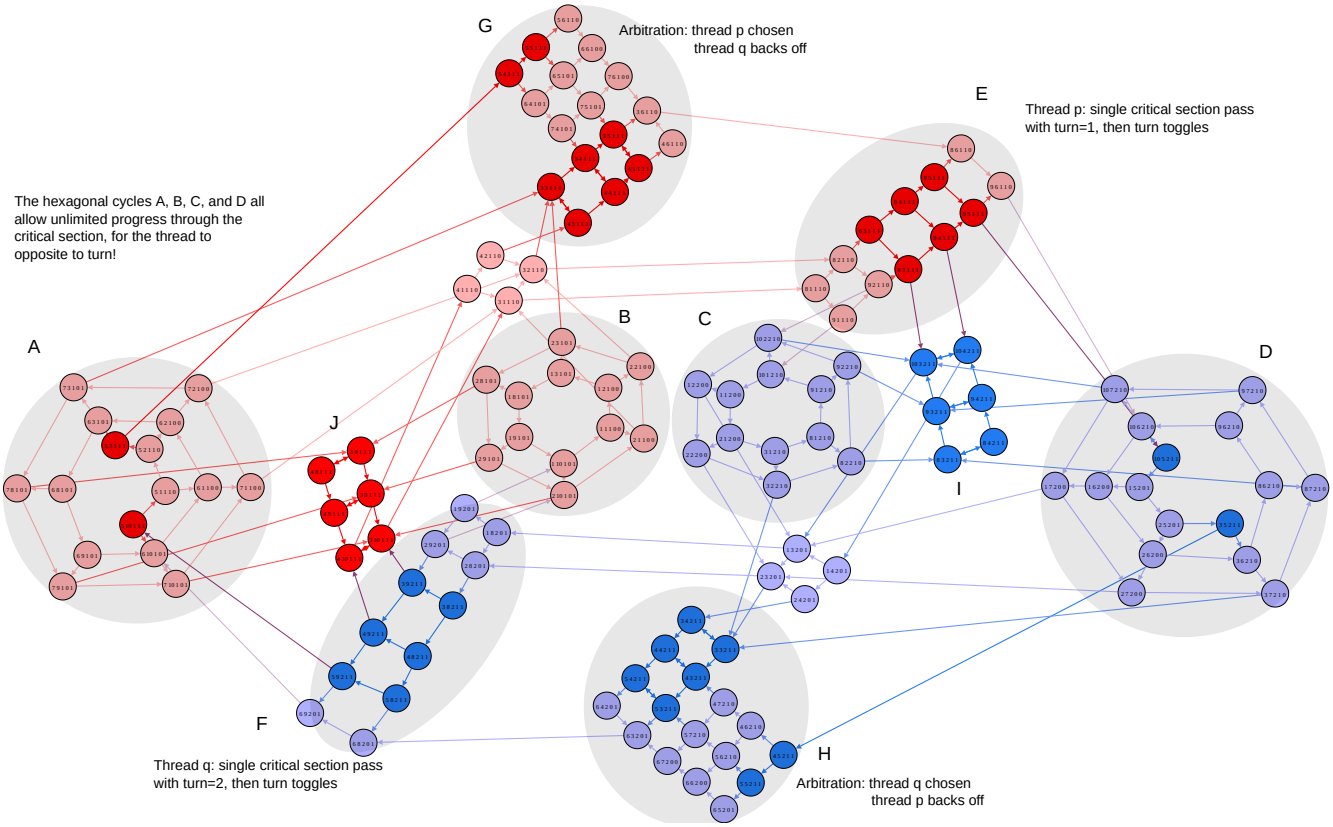


We can now see some structure here, but much of the logical structure is lost to the eye. Let’s now go further and factorise the graph by considering broad ‘*super states*’ of a thread’s program graph: non-critical section, spin busy-wait, wait sleep and critical section. The Cartesian product of these ‘super states’ for two threads has then sixteen super states. We wish also to prove that these are not all reachable, in particular the safety property of (critical section(P), critical section(Q)).

Below I have regenerated the above graph but factorized into modular subgraphs:



It's useful to mark the individual modules and add annotations, which I'll do as follows. The remaining small unmarked modules are the branching steps p3.4 and q3.4 of the program graphs.



We can now clearly see the following:

- The turn=1 (red) subgraph is isomorphic to the turn=2 (blue) subgraph (this is easy to check).
- There are *six* regions of progress, three for each `turn` subgraph:
    1. Linear regions E and F: A single pass through the CS when it is its turn, then turn toggles.

    2. middle loops B and C: p progresses whilst q is at q1.2. Here p may make an unlimited number of loops. And vice versa.
    3. outer loops A and D: p progresses whilst q is waiting in p6.7. Here p may make an unlimited number of loops. And vice versa.
- There are two regions of contention arbitration G and H, one for each `turn` subgraph. Here the only exit is for the proces whose turn it is.
- There are two 'benign' spin lock regions, which merely spin until the other thread exits the CS.

Validation of Dekker's algorithm

From the graph we are able to prove the following system properties:

1. **Mutual exclusion**. No path to (CSp,CSq) exists. This means in our case (p8,q8) is not reachable, it is not present in our transition system graph.
2. **Liveness**. No program deadlock. A thread is always able to proceed, we never reach a terminal state (other than all entering non-CS). We see this from the fact that regions without a spin lock must progress without intra-region loops; and spin locks must eventually take the other non-looping path by fairness (the scheduler must eventually grant the other thread time).
3. **Starvation-free**. For any given process: if it reaches its CS pre-protocol (signalling its intention to enter the CS), then it will eventually proceed. Here we must check all branch points p3 do eventually lead to p8 (by symmetry this applies to q). However, on a basic level, note that all states in the graph are reachable and remain so due to its cyclic nature. Again Dekker's algorithm relies on *strong fairness*to ensure it will eventually take the alternative branch for spin loops and out of the unlimited CS progress loops of the other thread.

Conclusion

We saw that Dekker's algorithm while quick to state, actually resulted in a complex system with a very large number of states compared to its program graph. Starting with 10 program graph 'states' for a single thread it generated over 150 transition system states for two threads. It was this hidden complexity that made the early analysis of concurrent algorithms so challenging. Fortunately, modern methods based on *transition systems*, which fully characterize program behaviour, give us a precise tool for proof. This article showed how the system could be factored into independent modules that allowed us to reason about correctness of the vital properties of *mutual exclusion*, *liveness*, and *freedom from starvation*. These are exactly the techniques with which we reason about multithreaded programs and more recently lock-free programs. After all, we don't have a lock in Dekker's algorithm!