Correctness of Hierarchical MCS Locks with Timeout

HMCS-T is a very involved protocol. The system is stateful; the values of prior acquisition efforts affect the subsequent acquisition efforts. Also, the status of successors, predecessors, ancestors, and descendants affect steps followed by the protocol. The ability to make the protocol fully non-blocking leads to modifications to the `next` field, which causes deviation from the original MCS lock protocol both in acquisition and release. At several places, unconditional field updates are replaced with `SWAP` or `CAS` operations.

We follow a multi-step approach to prove the correctness of HMCS-T. To demonstrate the correctness of HMCS-T lock, we make use of the Spin [1] model checking. Model checking causes a combinatorial explosion even to simulate a handful of threads. First we understand the minimal, sufficient configurations necessary to prove safety properties of a single level of lock in the tree. We construct HMCS-T locks that represent these configurations. We model check these configurations, which proves the correctness of components of an HMCS-T lock. Finally, building upon these facts, we argue logically for the correctness of HMCS-T$\langle n \rangle$.

1. MINIMAL CONFIGURATION

We need to answer the following questions to design an HMCS-T lock configuration that is sufficient to exercise all possible thread interleaving in any arrangement:

- How many threads are sufficient?
- How many lock levels are sufficient?
- How many lock acquisitions per participant are sufficient?

To answer these questions, we build non-deterministic finite acceptors (NFAs) that capture the state transition for each shared variable. The shared variables are the `status` and `next` fields of a `QNode` and the `tail pointer` variable. The transitions of the status flag of a root-level `QNode` are different from the transitions of the status field of a non-root-level `QNode`. Figure 1 and Figure 2, respectively, show the NFA for the `status` field of a root-level and a non-root-level `QNode`. Figure 3 shows the NFA for the `next` field of any `QNode`. The `tail pointer` variable can be either `null` or non-`null`, and it is less interesting in designing the HMCS-T verification configurations. Appendix A, B, and C describe the transition associated with every edge shown in Figure 1, 2, and 3, respectively.

Node labels in Figure 1-2 represent the field values in those states, and the subscripts distinguish the same values that bear different meanings in different contexts. Solid black edges represent the actions taken by a thread t owning the `QNode` under scrutiny. Dotted blue edges represent the actions taken by a predecessor p of t. Dotted red edges represent the actions taken by a successor s of t. Thick black edges represent beginning of a new acquisition effort by a thread t that owns the `QNode`. Any subsequent path formed only of solid black edges represents a sequence of actions taken by a same thread of execution. Since the first operation in any acquisition is `SWAP`ing the status field, every new acquisition edge has a $\texttt{W}_i$ node as its sink. Green color filled node(s) represent the state(s) where the lock contending thread t has become the owner of the lock at that level.

The NFA provides the following key insights:

1. **Three participants:** Any edge can be traversed via a path starting at the start state that involves no more than a predecessor (dotted blue edge), self (black edge), and a successor (dotted red edge) in Figures 1, 2, and 3. Hence, three participants (a predecessor, self, and a successor) are sufficient to exercise all possible transitions that the status field of a `QNode` may go through.
2. **Two rounds:** Any edge can be traversed via a path starting at the start state that involves no more than two "begin acquisition" (thick black line) edges. Hence, two rounds of acquisitions on the same `QNode` are sufficient to exercise all possible transitions. This means, at least, one thread should try two acquisitions. The other two threads can perform one acquisition each to exercise all interleaving of the third thread that performs two acquisitions.
3. **Three levels:** The edge $C_1 \rightarrow W_4$ in Figure 2 demands that a thread t_1 to have acquired the lock at the current node `q` at level l and abandoned at an ancestor level and a different thread t_2, a peer of t_1 at a level $< l$, to have inherited the level l lock from t_1. Hence, there should, at least, be two threads at level $l-1$, which can cause one of them (say t_1) to acquire locks at level $l-1$ and l but timeout at level $l+1$ and eventually grant the locks at level $l-1$ and l to another thread (say t_2). Three levels, parent, current, and children are sufficient to exercise all possible transitions in a non-root-level `QNode` .

To elaborate on Property 1 and 2, we describe a few interesting transitions in Figure 1. The edge $U_2 \rightarrow U_3$ needs t to have a predecessor to reach $\texttt{U}_2$ and then a successor to cause impatience during the release protocol to transition to $\texttt{U}_3$. The edge $W_3 \rightarrow R_2$ needs t to have a predecessor to reach $\texttt{U}_2$ and then the second round of acquisition attempt by t to reach $\texttt{W}_3$ and then a successor to make t impatient in

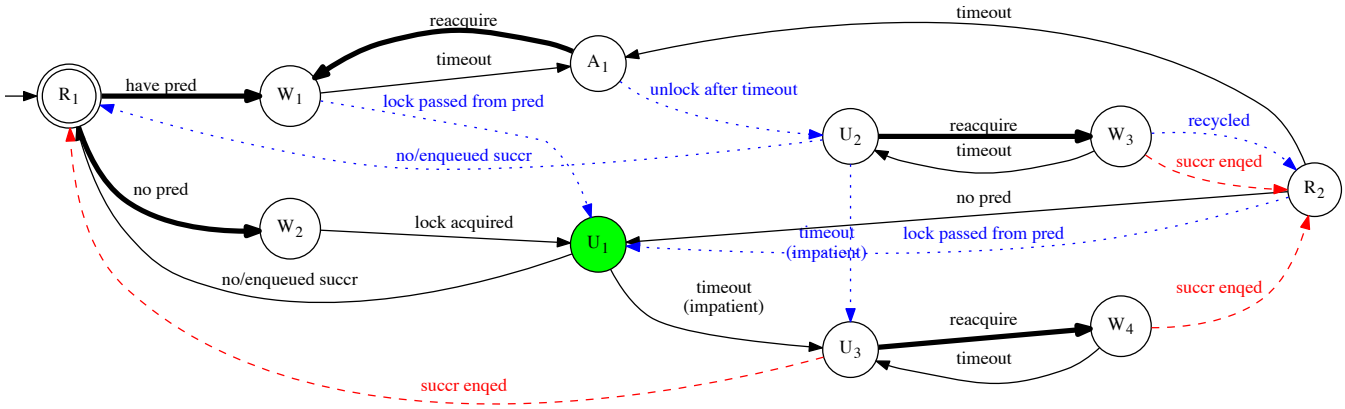

Figure 1: NFA for a `QNode` status field in HMCS-T$\langle 1 \rangle$.

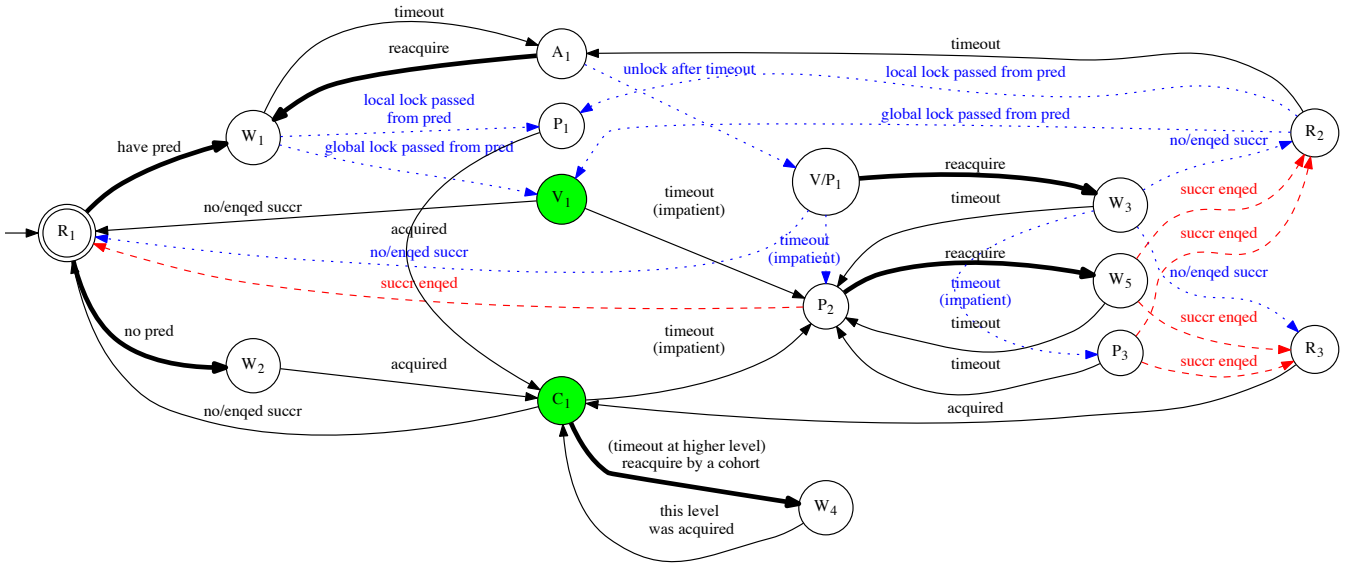

Figure 2: NFA for the status field of a non-root-level `QNode`.

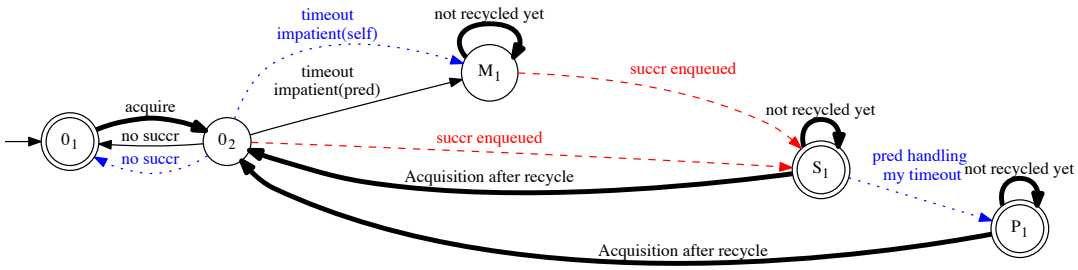

Figure 3: NFA for a `QNode next` field in HMCS-T$\langle n \rangle$. There is no designated "lock acquired" node.

Figure 4: Legend for Figures 1, 2, and 3

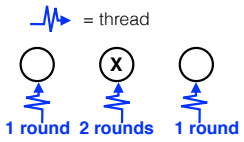

Figure 5: Model checking configuration to exercise all possible interleaving for a thread at root level.

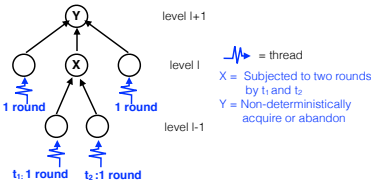

Figure 6: Model checking configuration to exercise all possible interleaving for a thread at a non-root level.

its release protocol to eventually make the successor update t's status to $\texttt{R}_2$. The edge $R_2 \rightarrow U_1$ and edge $R_2 \rightarrow A_1$ need at least two rounds of acquisitions by t and a successor s to reach $\texttt{R}_2$. The same successor s can act as a predecessor for edge $R_2 \rightarrow U_1$ transition. Similarly, s can act as a predecessor leading to a timeout to cause edge $R_2 \rightarrow A_1$ transition.

NFAs, unfortunately, do not capture an important safety property—mutual exclusion. An NFA is ill-defined if the ownership of a `QNode` is not exclusive, which can happen if another thread belonging to the same domain starts modifying a shared `QNode`. To check the mutual exclusion property, we exercise all possible thread interleaving in a model checking phase.

To exercise all states of the root-level lock we use a thread configuration shown in Figure 5. The thread under scrutiny will be subjected to two rounds of acquisitions and the other two threads perform one round of acquisition each. Since model checking will exercise all interleaving, the timeout value is immaterial.

To exercise all states of the non-root-level lock, we use a thread configuration shown in Figure 6. There are two threads at level 1, which can causes one of them (say t_1) to acquire the locks at level 1 and 2 but timeout at level 3 and eventually grant the ownership of locks at level 1 and 2 to another thread (say t_2). The presence of two threads at level 1, also causes the common ancestor X, the `QNode` under scrutiny at level l, to go through the necessary two rounds of acquisitions. The other two participants—a successor s, and a predecessor p at level l— perform only one round of acquisition each. The model checking does not require s and p to begin the protocol at the leaf level, which avoids exercising some non-interesting interleavings. Hence, we set up s and p without children. Note that such arrangement is for model checking only; the HMCS-T lock admits new acquisitions starting at the leaf level only. In total, we need 4 threads, 2 at level 1 sharing the parent X, and 3 (of which one would have ascended from 1) at level 2. The behavior at level 3 will be non-deterministic—either a successful acquisition or abandonment to simulate all possible transitions in X. Non-deterministic behavior is easy to exhibit in Spin [1].

The verification checks for the assertion that two threads are never simultaneously in the critical section for the configuration in Figure 5. This assertion ensures that the root-level lock ensures mutual exclusion to the critical section if each `QNode` is accessed by descendent threads in a mutually exclusive manner. For the configuration in Figure 6, we check that t_1 and t_2 never simultaneously acquire the level $l-1$ lock and no two threads ever simultaneously acquire the level l lock. This assertion ensures that a non-root-level lock ensures mutual exclusion to its next level if each `QNode` is accessed by descendent threads in a mutually exclusive manner.

Additionally, the NFAs in Figure 1, 2, and 3 provide insights into the following key properties:

1. **Livelock Freedom:** There does not exist any cycle without at least one "begin new acquisition" edge. Hence, there cannot be perpetual state transitions (live lock) without user opting to start another round of lock acquisition.

2. **Starvation Freedom:** Every $\texttt{W}_i$ node (beginning of a new acquisition) has a path to the lock owning state ($\texttt{U}_1$ in Figure 1 and $\texttt{V}_1$ and $\texttt{C}_1$ in Figure 2), if it is not allowed to traverse any *timeout* edge. This implies, every thread that starts its acquisition process and does not timeout, eventually acquires the lock. The `next` field does not decide the lock ownership and hence ignored.

3. **Bounded Steps to Release:** There exists a finite-length solid-black edge path from lock owner state to another node η such that a new acquisition (thick black edge) effort can begin at η. This implies, 1) an acquired lock can be released in a bounded number of steps by the lock owner and 2) once the lock is released, the `QNode` can be subjected to another acquisition attempt immediately.

4. **Bounded Steps on Timeout:** Every node that is *not* source node of a new acquisition edge (thick black edge) has a solid-black edge path to the source of a timeout edge. This implies that in any state after starting an acquisition process if a timeout occurs, t can abandon the protocol in a bounded number of steps. Source nodes of new acquisition edges are precluded because one cannot start an abandonment without having started an acquisition.

5. **Deadlock Freedom:** Every node has a path (there is an ϵ path to itself) formed out of solid-black edges to a node from where a new acquisition can begin.

2. CORRECTNESS OF HMCS-T$\langle N \rangle$

To establish the mutual exclusion guarantee of HMCS-T$\langle n \rangle$, we take the following steps:

Lemma 2.1 (Root level lock ensures mutual exclusion:) *A root-level lock ensures mutual exclusion if every root-level `QNode` is owned by a descendent in a mutually exclusive manner.*

Proof. Verified by model checking a root-level lock with the configuration shown in Figure 5. □

Lemma 2.2 (Non-root level lock ensures mutual exclusion:) *A non-root-level lock admits mutually exclusive access to the next level lock if every `QNode` at that level is owned by a single descendent at a time.*

Proof. Verified by model checking a non-root-level in an HMCS-T lock with the configuration shown in Figure 6. □

Fact 2.1 (Exclusive ownership of leaf-level node:) *Every `QNode` at leaf level is owned by a unique thread, and the ownership is never shared with any other thread.*

Theorem 2.1 (HMCS-T ensures mutual exclusion:) *HMCS-T$\langle n \rangle$ ensures mutual exclusion to the critical section it protects.*

PROOF. HMCS-T$\langle n\rangle$ is composed of a root-level lock and $n-1$ non-root-level locks. Each level ensures mutual exclusion to the level above as long the threads from descendent levels (if any) accesses the shared `QNode` at the current level in a mutually exclusive manner. Assume HMCS-T$\langle n\rangle$ does not ensure mutual exclusion to the critical section. This means two threads t_1 and t_2 can simultaneously be in the critical section. Both t_1 and t_2 are either 1) peers at level n and hence compete for the root-level lock at level n, or 2) belong to the same domain and hence compete for a non-root-level lock at a level $l < n$.

If t_1 and t_2 are peers at level n, they will enqueue, two different `QNode`s and compete for the root-level lock and by Lemma 2.1 only one of them can be in the critical section at a time. Hence, t_1 and t_2 cannot be peers at the root-level.

Now, t_1 and t_2 are either peers at level $n-1$ or belong to the same domain at level $l' < n-1$. If t_1 and t_2 are peers at level $n-1$, they will enqueue two different `QNode`s and compete for the non-root-level lock at level $n-1$ and by Lemma 2.2 only one of them can own the level $n-1$ lock ensuring the mutual exclusion between them. Hence, t_1 and t_2 cannot be peers at level $n-1$.

Since HMCS-T$\langle n\rangle$ has only a finite number of levels, by extrapolation, t_1 and t_2 are either peers at the leaf level or share the same `QNode` at the leaf level. If t_1 and t_2 are peers at the leaf level, they will enqueue two different `QNode`s and compete for the non-root-level lock at the leaf level and by Fact 2.2 only one of them can own the leaf level lock ensuring the mutual exclusion between them. Hence, t_1 and t_2 must be sharing the same `QNode` at the leaf level. By Lemma 2.1, no two threads can share the same `QNode` at the leaf level, hence $t_1 = t_2$, which contradicts the assumption.

Hence, only one thread can be in the critical section in HMCS-T$\langle n\rangle$. □

The desirable attributes—starvation freedom, live-lock and deadlock freedom, bounded steps to release or time out—for a given level of lock do not translate to the same for an entire HMCS-T$\langle n\rangle$ lock. To establish these properties for HMCS$\langle n\rangle$, we make the following claims:

Fact 2.2 (Ordered acquisition:) *Any thread in HMCS-T lock of n levels obeys a monotonically increasing order in acquisition effort starting from level* 1 *and ending at level* $l \leq n$.

Fact 2.3 (Ordered release and abandonment:) *HMCS-T lock of n levels obeys a bitonically ordered release and abandonment—monotonically increasing in level followed by monotonically decreasing in level. A thread owning locks* 1 <= ***prefix:suffix*** $\leq n$ *either releases the* ***suffix*** *locks before releasing the ownership of remaining* ***prefix*** *locks or delegates the same responsibility to another thread that becomes the owner of entire* ***prefix:suffix*** *locks.*

Theorem 2.2 *HMCS-T$\langle n\rangle$ guarantees live-lock freedom, deadlock freedom, starvation freedom, bounded steps to release, and bounded steps on timeout.*

PROOF. HMCS-T$\langle n\rangle$ is composed of a root-level lock and $n-1$ non-root-level locks. By Fact 2.2 and 2.3, every thread follows an ordered acquisition and release or abandonment protocol. Hence, each thread goes through a finite number of levels in any process. At each level, root or non-root, the NFA that a thread is subjected to for its `QNode`, ensures live-lock freedom, deadlock freedom, starvation freedom, bounded steps to release, and bounded steps on timeout if the `QNode` is accessed mutually exclusively by descendants that share the same ancestor `QNode`. By Theorem 2.1, each `QNode` is owned by a descendent thread in a mutually exclusive manner. Hence, by construction HMCS-T$\langle n\rangle$ ensures live-lock freedom, deadlock freedom, starvation freedom, bounded steps to release, and bounded steps on timeout. □

APPENDIX

A. NFA FOR THE STATUS FIELD OF A ROOT-LEVEL QNODE

The status always starts in $\mathtt{R}_1$ state. **All other states are transient; a correctly implemented HMCS-T$\langle 1\rangle$ ought to revert the status of very `QNode` to $\mathtt{R}_1$ eventually.** On a fresh acquisition in the $\mathtt{R}_1$ state of a `QNode q`, the initial `SWAP` on `q.status` moves it non-deterministically to either $\mathtt{W}_1$ (if there was a predecessor) or $\mathtt{W}_2$ (no predecessor).

If no predecessor, the thread t updates `q.status` to $\mathtt{U}_1$ (edge $W_2 \rightarrow U_1$). In $\mathtt{U}_1$, if t has a successor s that has already advertised itself with `q.next` or there is no successor, t releases the lock and updates `q.status` to $\mathtt{R}_1$ (edge $U_1 \rightarrow R_1$). In $\mathtt{U}_2$, if t leaves due to timeout because a successor s has not updated `q.next`, the NFA transitions into state $\mathtt{U}_3$ (edge $U_1 \rightarrow U_3$). In $\mathtt{U}_3$, if s advertises itself and recycles `q.status`, the NFA transitions to $\mathtt{R}_1$ (edge $U_3 \rightarrow R_1$). In $\mathtt{U}_3$, if t attempts to re-acquire the lock, it will `SWAP q.status` to $\mathtt{W}_4$ (edge $U_3 \rightarrow W_4$). If t times out in $\mathtt{W}_4$ while waiting for it to become $\mathtt{R}$, it reverts the state back to $\mathtt{U}_3$ (edge $W_4 \rightarrow U_3$). In $\mathtt{W}_4$, if s advertises itself and recycles `q.status`, the NFA transitions to $\mathtt{R}_2$ (edge $W_4 \rightarrow R_2$).

In $\mathtt{W}_1$, a predecessor may pass the lock to the waiting thread t updating `q.status` to $\mathtt{U}_1$ (edge $W_1 \rightarrow U_1$). If t times out in $\mathtt{W}_1$, it updates the state to $\mathtt{A}_1$ (edge $W_1 \rightarrow A_1$). In $\mathtt{A}_1$, a predecessor p may move the status to $\mathtt{U}_2$ (edge $A_1 \rightarrow U_2$). In $\mathtt{A}_1$, any attempt by t to re-acquire the lock reverts the state to $\mathtt{W}_1$ (edge $A_1 \rightarrow W_1$). In $\mathtt{U}_2$, if p manages to successfully release the lock, it will eventually transition `q.status` to $\mathtt{R}_1$ (edge $U_2 \rightarrow R_1$). In $\mathtt{U}_2$, if p times out (impatient) waiting for a successor delayed in updating `q.next` field, the NFA transitions to $\mathtt{U}_3$ (edge $U_2 \rightarrow U_3$). In $\mathtt{U}_2$, any attempt by t to re-acquire the lock moves the state to $\mathtt{W}_3$ (edge $U_2 \rightarrow W_3$). If t times out in $\mathtt{W}_3$, it reverts the state to $\mathtt{U}_2$ (edge $W_3 \rightarrow U_2$). In $\mathtt{W}_3$, either a predecessor may update the state to recycled $\mathtt{R}_2$, or an impatient predecessor may time out and a successor may update the state to recycled $\mathtt{R}_2$ (edge $W_3 \rightarrow R_2$).

In $\mathtt{R}_2$, t will reenqueue the `QNode` and it may acquire the lock via transition to $\mathtt{U}_1$ either because it has no predecessors or a predecessor passed the lock (edge $R_2 \rightarrow U_1$). In $\mathtt{R}_2$, after enqueuing the node, if t times out waiting for the lock, it will transition to $\mathtt{A}_1$ (edge $R_2 \rightarrow A_1$).

B. NFA FOR THE STATUS FIELD OF A NON-ROOT-LEVEL QNODE

We now describe the state diagram for the `status` field of a non-root-level `QNode`.

The status always starts in $\mathtt{R}_1$ state. **All other states are transient, a correctly implemented non-root-level ought to revert the status of very `QNode` to $\mathtt{R}_1$ eventually.** On a fresh acquisition in the $\mathtt{R}_1$ state of a `QNode q`, the initial `SWAP` on `q.status` moves it non-deterministically to either $\mathtt{W}_1$ (if there was a predecessor) or $\mathtt{W}_2$ (no predecessor).

If no predecessor, the thread t updates `q.status` to $\texttt{C}_1$ (edge $W_2 \rightarrow C_1$). IN $\texttt{C}_1$, if t has a successor s that has already advertised itself with `q.next` or there is no successor, t releases the lock and updates `q.status` to $\texttt{R}_1$ (edge $C_1 \rightarrow R_1$). In $\texttt{C}_1$, if t leaves due to timeout because a successor s has not updated `q.next`, t leaves `q` by updating its status to $\texttt{P}_2$ (edge $C_1 \rightarrow P_2$). In $\texttt{P}_2$, if s advertises itself and recycles `q.status`, the NFA transitions to $\texttt{R}_1$ (edge $P_2 \rightarrow R_1$). In $\texttt{P}_2$, if t attempts to re-acquire the lock, it will `SWAP` `q.status` to $\texttt{W}_5$ (edge $P_2 \rightarrow W_5$). If t times out in $\texttt{W}_5$ while waiting for it to become `R`, it reverts the state back to $\texttt{P}_2$ (edge $W_5 \rightarrow P_2$). In $\texttt{W}_5$, if s advertises itself and recycles `q.status`, the NFA non-deterministically transitions to either $\texttt{R}_3$ (edge $W_5 \rightarrow R_3$, if it finds no predecessor by the time t re-enqueues the node) or to $\texttt{R}_2$ (edge $W_5 \rightarrow R_2$, if a predecessor is present by the time t re-enqueued the node). In $\texttt{R}_3$, t will acquire the lock immediately and update the status to $\texttt{C}_1$ (edge $R_3 \rightarrow C_1$).

In $\texttt{C}_1$, having acquired the current level (say l) lock t may ascend to an ancestor level and it may abandon the lock at that level. In an effort to release the locks already held, t may pass its locks including l lock to another thread, say t_2. When t_2 begins its acquisition process at level l, it will `SWAP` `q.status` to $\texttt{W}_4$ (edge $C_1 \rightarrow W_4$) and immediately realize that it inherited this lock and revert `q.status` to $\texttt{C}_1$ (edge $W_4 \rightarrow C_1$)

If t times out in $\texttt{W}_1$, it updates the state to $\texttt{A}_1$ (edge $W_1 \rightarrow A_1$). In $\texttt{A}_1$, a predecessor p may attempt to pass all locks it holds (`V`, a legal lock passing value) or only a prefix of locks (`P`) (edge $A_1 \rightarrow V/P_1$). In $\texttt{A}_1$, any attempt by t to re-acquire the lock reverts the state to $\texttt{W}_1$ (edge $A_1 \rightarrow W_1$). In $\texttt{V/P}_1$, if p manages to successfully release the lock, it will eventually transition `q.status` to $\texttt{R}_1$ (edge $V/P_1 \rightarrow R_1$). In $\texttt{V/P}_1$, if p times out (impatient) waiting for a successor delayed in updating `q.next` field, the NFA transitions to $\texttt{P}_2$ (edge $V/P_1 \rightarrow P_2$).

In $\texttt{W}_1$, a predecessor may pass the global lock (all locks on path to the root) to t by updating `q.status` to a legal passing value $\texttt{V}_1$ (edge $W_1 \rightarrow V_1$). In $\texttt{V}_1$, if t has a successor s that has already advertised itself with `q.next` or there is no successor, t releases the lock and updates `q.status` to $\texttt{R}_1$ (edge $V_1 \rightarrow R_1$). In $\texttt{V}_1$, if t leaves due to timeout because a successor s has not updated `q.next`, t would have already released all ancestral locks and then it leaves `q` by updating `q.status` to $\texttt{P}_2$ (edge $V_1 \rightarrow P_2$). In $\texttt{W}_1$, a predecessor may pass only the local lock (having already released all its ancestral locks) to t by updating `q.status` to $\texttt{P}_1$ (edge $W_1 \rightarrow P_1$). IN $\texttt{P}_1$, when t notices that it owns the lock at that level, it will update the status to $\texttt{C}_1$ to indicate the beginning of a new cohort (edge $P_1 \rightarrow C_1$).

In $\texttt{V/P}_1$, t may attempt to re-acquire the lock, which transitions it to $\texttt{W}_3$ (edge $V/P_1 \rightarrow W_2$). In this state, t will have to wait till the node is recycled. If t times out while waiting for the status to become `R` in $\texttt{W}_3$, it will update the status to $\texttt{P}_2$ and leave (edge $W_3 \rightarrow P_2$). In $\texttt{W}_3$, if the predecessor p trying to pass the lock becomes impatient because a successor s has not updated `q.next`, p leaves `q` by updating its status to $\texttt{P}_3$ (edge $W_3 \rightarrow P_3$). If t times out while waiting for the status to become `R` in $\texttt{P}_3$, it will update the status to $\texttt{P}_2$ and leave (edge $P_3 \rightarrow P_2$). In $\texttt{P}_3$, if s advertises itself and recycles `q.status`, the NFA non-deterministically transitions to either $\texttt{R}_3$ (edge $P_3 \rightarrow R_3$, if it finds no predecessor by the time t re-enqueues the node) or to $\texttt{R}_2$ (edge $P_3 \rightarrow R_2$, if a predecessor is present by the time t re-enqueued the node).

In $\texttt{W}_3$, if the predecessor p manages to successfully release the lock to some other thread or relinquish the lock, p it will eventually transition `q.status` to $\texttt{R}_3$ (edge $W_3 \rightarrow R_3$, if t finds no predecessor by the time it re-enqueues the node) or to $\texttt{R}_2$ (edge $W_3 \rightarrow R_2$, if a predecessor is present by the time t re-enqueues the node).

In $\texttt{R}_2$, t will reenqueue the `QNode` and it may inherit the global lock (transition to $\texttt{V}_1$, edge $R_2 \rightarrow V_1$) or inherit only lock prefix (transition to $\texttt{P}_1$, edge $R_2 \rightarrow P_1$) from one of its predecessors. In $\texttt{R}_2$, t may timeout and abandon while waiting for the lock (edge $R_2 \rightarrow A_1$).

C. NFA FOR THE NEXT FIELD OF A QNODE

We now describe the state diagram for the `next` field. The `next` field starts with a `null` value in state 0_1. At the beginning of an acquisition, thread t transitions to 0_2, where the value of the `next` field remains unchanged from before (edge $0_1 \rightarrow 0_2$). If t finishes relinquishing the lock, the state reverts to 0_1 (edge $0_2 \rightarrow 0_1$). This transition can happen either by t itself (black solid edge) or after t has abandoned, which case a predecessor may act on t's behalf (blue colored dotted edge).

If a successor enqueues and advertises itself with a legal `QNode` pointer value `S`, NFA transitions to S_1 (edge $0_2 \rightarrow S_1$). t may successfully acquire the lock and release, which leaves it in S_1. t may timeout and abandon, which leaves it in S_1 and subsequent attempts to acquire by t will leave it in S_1 until a predecessor marks the `QNode` for recycling at which point t resets the next pointer to `null` just before enqueuing (edge $S_1 \rightarrow 0_2$). In S_1, if t times out, a predecessor, may reuse the `next` field to remember the predecessor on its forward journey to find a waiting successor (edge $S_1 \rightarrow P_1$). In S_1, if t attempts to re-acquire, it will wait and possibly timeout (edge $S_1 \rightarrow S_1$). In P_1, once a predecessor has recycled the `QNode`, t will reset the next pointer to `null` and re-enqueue (edge $P_1 \rightarrow 0_2$). In P_1, if t attempts to re-acquire, it will wait and possibly timeout (edge $P_1 \rightarrow P_1$). In 0_2, if t timeouts during release waiting for the successor to update the `next` pointer, t writes M_1 (edge $0_2 \rightarrow M_1$). If t times out during acquire in 0_2, a predecessor may trigger the edge $0_2 \rightarrow M_1$ transition. In M_1, if t attempts to re-acquire, it will wait and possibly timeout (edge $M_1 \rightarrow M_1$) until the node is recycled by the successor (edge $M_1 \rightarrow S_1$).

D. REFERENCES

[1] G. J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.