# The Beehive Ring and Multiring Lock Protocols

Thomas L. Rodeheffer
Microsoft Research, Silicon Valley

November 2, 2010

Beehive is an experimental many-core computer implemented on a single FPGA. The system includes a number of RISC cores that are connected to each other and to main memory using a token ring interconnect. Among other things, the token ring is used to maintain a set of global hardware locks using a *ring lock protocol*. A multiring extension of the Beehive token ring has been proposed for which a more complicated *multiring lock protocol* is required.

This paper describes the ring lock protocol and the multiring lock protocol, presents correctness proofs, and discusses the results of checking their formal specifications using the TLC model checker. The formal specifications are listed in appendices.

## 1 Introduction

Beehive [2, 3] is an experimental many-core computer implemented on a single FPGA. The system includes a number of RISC cores that are connected to each other and to main memory using a token ring interconnect.

In implementing a many-core computer on a single FPGA, we found that the limiting resource was long wires. Using a token ring for the interconnect made all the inter-core wiring local, and hence routable.

The token ring is used for main memory access, core-to-core messages, and to maintain global hardware locks. In this paper we focus on the *ring link protocol* which is used to maintain the locks. The ring lock protocol is implemented in the current Beehive hardware.

The basic ideas of the ring lock protocol are (1) each node keeps track of the locks it holds and (2) when a node wishes to acquire an additional lock it first sends a message around the ring to make sure no other node has it. This can be delicate if several nodes attempt to acquire the same lock at the same time.

Although the Beehive token ring provides a convenient mechanism for serialization and global enquiry, the latency increases linearly as the number of nodes increases. Even though the current FPGA implementations of Beehive are restricted to fewer than sixteen nodes due resource limitations, it would be desirable to have a design which scaled better in latency. For this purpose, a multiring extension of the Beehive token ring has been proposed, in which a large token ring is implemented as a ring of subrings. Each subring is managed by a junction node. The junction node either routes a train around the subring or, when none of the nodes on the subring need access to the train, sends the train by as a bypass train. Ideally, most traffic bypasses the subrings. The multiring extension can be applied recursively, producing a hierarchical multiring structure in which latency scales as the logarithm of the total number of nodes.

The multiring extension uses a *multiring lock protocol* to maintain the global hardware locks. The multiring lock protocol is based on the ring lock protocol, but the ability of a junction node to permit traffic to bypass its subring creates a difficulty. The main idea in the multiring lock protocol is to arrange for a junction node to act on behalf of the user nodes on its subring.

We prove that both the ring lock protocol and the multiring lock protocol satisfy the following properties:

**Mutual exclusion (safety):** At no time can two different nodes hold the same lock.

**Lock acquisition (liveness):** Whenever a lock is desired infinitely often, eventually the lock is held.

Mutual exclusion is a safety propery that any locking protocol must satisfy. Lock acquisition is a liveness property that ensures progress.

The remainder of this paper is organized as follows. Section 2 describes the Beehive token ring. Section 3 describes the ring lock protocol, proves its safety and liveness, and presents results from model checking a formal specification of the protocol. Section 4 describes the multiring extension. Section 5 describes the multiring lock protocol, proves its safety and liveness, and presents results from model checking a formal specification of the protocol. Section 6 concludes. Listings of the formal specifications are contained in Appendix A and Appendix B.

## 2 The Beehive token ring

Figure 1 shows the basic idea of the Beehive token ring. The ring is composed of a *zero node*, which manages the ring, and a number of *user nodes*. Activity on the ring is conceived as a *train* that starts from the zero node, chugs one car per time slot through each of the user nodes, and then ends up back at the zero node. A train consists of a *token* car (its engine) followed by a number of data cars of various types.

In the Beehive hardware implementation, the *token* contains a field that explicitly denotes the number of data cars following, but we adopt a simpler model in which the end of the train is denoted by an *idle* car. Additional *idle* cars follow as needed to fill time slots before the start of the next train. In Beehive, the zero node also serves as the main memory controller. Each RISC core is a user node on the ring.

A user node sees the cars one at a time. It can rewrite a data car (substituting its own contents for the former contents of the data car) and it can add new data cars to the end of the train. Adding a data car is the same as rewriting the first *idle* car that follows the train. A user node rewrites a data car, for example, to erase its own request that circled the ring, as explained below.

Each car has a type and generally contains the node id of the originating node along with perhaps some additional information. In order for data to go all the way around the ring, the zero node recycles most types of cars that it receives from the current train to use as the initial cars of the next train. The *token*, for example, is always recycled. A *null* car, on the other hand, is never recycled. When a user node receives back a data car it sent, it rewrites the car with *null* so that no further node will be concerned with it and the zero node will discard it.

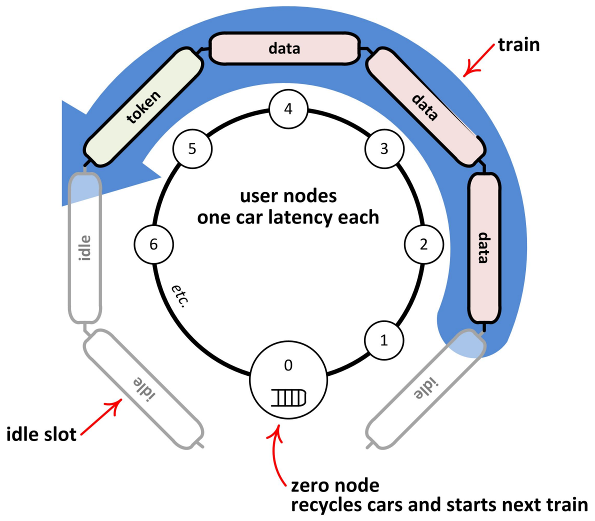

Figure 1: The Beehive token ring.

Only one train is permitted on the ring at a time. The zero node does not start the next train until it receives the end of the current train. Hence user nodes can add as many cars to the train as desired, up to the limit of buffering at the zero node.

In Beehive, the token ring is globally synchronous, with each car advancing to the next node on each time slot as controlled by a central clock. However, since nodes only communicate with their neighbors, it would be possible to forward the clock along the ring so that time slots occurred locally at each node with a possible phase delay. In such a design, the zero node would use its buffer to rephase incoming slots to the original clock. From the point of view of the protocols, everything would work exactly the same as if time slots were globally synchronous, so, for simplicity, we describe the system that way. In our description, time is represented as an integer.

## 3 The ring lock protocol

Beehive provides a small number of global hardware locks that are managed by the *ring lock protocol*. Each lock is identified to the hardware using a small, non-negative *lock index*. A lock index is a value in the range 0 through 63 for the current Beehive hardware. The meaning of each lock is assigned by software.

We nexr describe the ring lock protocol, prove its safety and liveness, and discuss how we checked the protocol using a formal specification.

## 3.1 Description

The *ring lock protocol* manages a set of global hardware locks. Each lock is identified by a lock index which is a small, non-negative integer.

For simplicity, the protocol permits only user nodes to acquire and release locks. The hardware in each node keeps track of which locks it holds using a small array of bits. Note that a lock does not have a concrete existence in any particular place. Instead, a lock is an abstract concept managed via the ring lock protocol.

Initially, no nodes hold any locks. There are three actions on locks that the protocol permits the software on a node to perform: (1) attempting to acquire a lock that is not yet held by the node, (2) releasing a lock that the node holds, and (3) forcing the release of a lock not held by the node, but possibly held by some other node in the system. The details of these actions are described below.

The ring lock protocol uses three special types of data cars: *reqp*, *failp*, and *dov*. Each of these cars contains the node id of the node that creates the car and the lock id of a particular lock of interest. The *reqp* car is used when a node attempts to acquire a lock; it is rewritten to *failp* if some other node already has the lock. The *dov* car is used to force the release of a lock. These cars circle the ring and are rewritten to *null* when they return to the node that created them.

**Acquiring a lock.** When software instructs a node $n_1$ to attempt to acquire a lock not yet held by the node, the node adds a *reqp* car to the train and waits for the car to circle the ring. Each node $n_2$ inspects the *reqp* car as it circles the ring, and, if $n_2$ holds the lock in question, it rewrites the type of the car to *failp*. Eventually, the car circles the ring and returns to $n_1$. If it returns as a *reqp*, then $n_1$ has acquired the lock, and it sets the corresponding lock bit in its array. If it returns as a *failp*, then the attempt to acquire the lock has failed. In either case, $n_1$ rewrites the car with a *null*. The software is informed of whether the attempt was a success or failure.

**Releasing a lock.** When software instructs a node to release a lock held by the node, the node merely clears the corresponding lock bit.

**Forcing the release of a lock.** When software instructs a node $n_1$ to force the release of a lock not held by the node, the node adds a *dov* car to the train and waits for the car to circle the ring. Each node inspects the *dov* car as it goes past and it clears the corresponding lock bit in its array if set. Eventually, the car circles the ring and returns to $n_1$, which rewrites it with a *null*.

## 3.2 Safety and liveness

We now proceed to prove that the ring lock protocol satisfies the mutual exclusion and lock acquisition properties. The proof of mutual exclusion is fairly subtle.

We define $\varphi_{c,t}$, the *odometer* of car $c$ at time $t$, as the number of forwarding steps car $c$ has experienced from its creation until time $t$. A car is created when some node first adds it to a train, at which time its odometer is 0. Each time the car is forwarded from one node to the next (whether or not it is rewritten), its odometer increases by 1. Since the zero node buffers cars between trains, the car may spend some time in the zero node while its odometer remains the same. When the car is recycled by the zero node and forwarded to the next node as part of the next train, its odometer again increases by 1.

Given two cars $c$ and $d$, we define $c \prec d$ to mean that car $c$ was first forwarded before car $d$. Since cars are only added to the end of a train, at any time step there can only be one car that is first forwarded during that time step. Hence $c \prec d$ means that if $c$ and $d$ ever appear on the same train, $c$ will appear ahead of $d$.

We show two important properties about the odometers of the cars in a train. Property 1 concerns all cars at a given time and Property 2 concerns all cars at a given node.

**Property 1: non-increasing odometer along a train at a given time.** Given two cars $c \prec d$ on the same train at time $t$, it follows that $\varphi_{c,t} \geq \varphi_{d,t}$. Furthermore, if $c$ and $d$ are buffered in different nodes at time $t$, it follows that $\varphi_{c,t} > \varphi_{d,t}$.

The proof is simple. Since $c \prec d$ and both cars exist on the same train at time $t$, it must be that $c$ is on the train ahead of $d$ at time $t$. Therefore, $d$ must have experienced a subsequence of the forwarding steps experienced by $c$, since the train pulls car $d$ along after wherever car $c$ has gone. Furthermore, this subsequence is proper if $c$ and $d$ are not buffered in the same node at time $t$. The desired result follows immediately.

This completes the proof of Property 1.

**Property 2: non-increasing odometer along a train at a given node.** Given two cars $c \prec d$ on the same train present in node $n$ at times $t_c$ and $t_d$, respectively, it follows that $\varphi_{c,t_c} \geq \varphi_{d,t_d}$.

The proof is simple. Observe that the train pulled car $c$ around from its creation until time $t_c$, when $c$ was buffered in node $n$. Similarly, the train pulled car $d$ around from its creation until time $t_d$, when likewise $d$ was buffered in node $n$. Since it is the same train for both $c$ and $d$, they both had to be pulled into node $n$ via exactly the same sequence of forwarding steps. Since $c \prec d$, it must therefore be the case that the sequence of forwarding steps experienced by $d$ from its creation until time $t_d$ is a suffix of the sequence of forwarding steps experienced by $c$ from its creation until time $t_c$. From this, the desired result follows immediately.

This completes the proof of Property 2.

Now we prove that the ring lock protocol guarantees mutual exclusion. In the ring lock protocol, user node $n_1$ claims a lock when its unchanged *reqp* car $c_1$ returns having circled the ring. Let $t_1$ be the time at which this happens and let $r$ be the number of nodes on the ring (including the zero node). Observe that $\varphi_{c_1,t_1} = r$.

By Property 1, no other node can have a car on the same train whose odometer at time $t_1$ is $r$. Since there is only one train at a time, no node other than $n_1$ can claim the same lock—or, indeed, any lock—at time $t_1$. So, in the ring lock protocol, we do not have to worry about multiple nodes claiming the same lock at the same time.

All that remains to be shown is that at time $t_1$ no user node $n_0 \neq n_1$ already held the lock. We will assume that such a node exists and derive a contradiction. Since by assumption $n_0$ already holds the lock at time $t_1$, there must be some most recent time $t_0 < t_1$ at which $n_0$ claims the lock. Observe that $n_0$ holds the lock continuously from $t_0$ through $t_1$. Let $c_0$ be the *reqp* car used by $n_0$ in claiming the lock at time $t_0$. Observe that $\varphi_{c_0,t_0} = r$.

Now, since there is only one train at a time, car $c_1$ is either added to the train sometime after time $t_0$ or it was already on the train at time $t_0$. Let us suppose it is added sometime after time $t_0$. Then it would have to completely circle the ring while $n_0$ holds the lock. But this would result in $n_0$ rewriting it as a *failp*, which is a contradiction.

Hence car $c_1$ must be on the train at time $t_0$. Since at time $t_0$ car $c_0$ is also on the train, car $c_1$ is either ahead of car $c_0$ or behind car $c_0$. Let us suppose it is ahead. Then since $n_0$ is a user node, $c_0$ and $c_1$ must be buffered in different nodes at time $t_0$ and by Property 1 we can conclude that $\varphi_{c_1,t_0} > r$. But $\varphi_{c_1,t_1} = r$, which implies $t_0 > t_1$, which is a contradiction.

Hence car $c_1$ must be behind car $c_0$. By Property 2, the odometer of car $c_1$ when it reaches node $n_0$ cannot exceed $r$. It cannot equal $r$ because that does not happen until car $c_1$ reaches node $n_1$ and by assumption $n_0 \neq n_1$. Therefore, car $c_1$ cannot have completed circling the ring until after having passed node $n_0$ while $n_0$ holds the lock. But this is a contradiction, because in such a case $n_0$ would have rewritten the car as a *failp*.

This completes the proof that the ring lock protocol guarantees mutual exclusion.

It is easy to establish that whenever one or more nodes attempt to acquire a lock, eventually some node holds the lock. If multiple user nodes attempt to acquire a lock that is not currently held by any node, precisely one node will manage to add its *reqp* car to the train first, this *reqp* will be the first to circle the ring, and consequently its node will acquire the lock. This is even stronger than the required lock acquisition property, which only requires that the lock eventually be held whenever it is desired infinitely often.

It would be nice to have an even stronger liveness property, for example: whenever a given node tries infinitely often to acquire a lock, eventually that node gets it. Unfortunately, the ring lock protocol does not satisfy this stronger liveness property.

## 3.3 Checking the protocol

Appendix A contains a formal specification of the ring lock protocol, written in TLA+ [1]. The specification starts out by defining various data types—the node identifier, the array of lock bits, the messages (cars), the activities that a node can be doing, the state of a node—and builds up to definitions of what a node does in a time slot as it receives the next message. These definitions are put

| user nodes | locks | runtime (sec) | depth | distinct states |
|---|---|---|---|---|
| 2 | 1 | 4 | 18 | 230 |
| 2 | 2 | 18 | 25 | 2062 |
| 3 | 1 | 16 | 24 | 1844 |
| 3 | 2 | 470 | 32 | 38644 |
| 4 | 1 | 141 | 29 | 12789 |
| 5 | 1 | 1408 | 35 | 83119 |
| 6 | 1 | 14873 | 40 | 504876 |

Table 1: Model checking results for the ring lock protocol specification. No violations were reported.

together into several next state relations: a complicated relation that advances a time step at all nodes simulateously and one relation for each of the three actions that the protocol permits a user node to perform. Then several invariants are defined, followed by temporal assumptions and temporal properties. Finally, everything is assembled into a single formula.

We used the TLC model checker to check several models of the ring lock protocol specification, with the results listed in Table 1. No violations of invariants or of temporal properties were reported. As is typical with model checking, the number of distinct states and the runtime of the model checker increase enormously as the model size increases, which makes checking feasible only for small models.

In order to gain confidence that model checking would actually find errors if the specification were buggy, it helps to intentionally introduce bugs and see if the model checker reports a violation. This is the purpose of the various *Bug...* definitions in the specification. A model configuration file can override these definitions to introduce the bugs.

We ran the TLC model checker on several such buggy configurations, described as follows:

**BugOmitCheckReqP** A node holding a lock omits to fail a *reqp* for that lock, resulting in the error that the lock may be claimed by more than one node at the same time.

**BugContinuousDoV** A node omits to process a *dov* it sent that circled the ring, resulting in the error that a request to force the release of a lock will never complete.

**BugOmitCheckDoV** A node holding a lock omits to release the lock when it receives a *dov* for that lock, resulting in the error that a request to force the release of a lock may fail to release the lock.

**BugOmitClaimLock** A node omits to claim a lock when its *reqp* returns after circling the ring, resulting in the error that a lock desired infinitely often may never be acquired.

Violations were reported in every case even with very small model sizes. Table 2 shows the results. The *trace length* is the number of states in the counterexample produced by TLC. An infinite trace is required to exhibit a counterexample of a temporal property; such a trace consits of an initial $i$ states followed by a cycle of $c$ states, which is listed as $i + c^*$. Even with the same model size, the depth and number of distinct states differ from one configuration to another because the buggy models act in different ways and, furthermore, TLC does not always need to explore the entire state space before it discovers a violation.

## 4 The multiring extension

Figure 2 shows the basic idea of the multiring extension. As in the normal Beehive token ring, the ring is managed by a *zero node*. However, one or more of the user nodes on the ring are replaced with *junction nodes*, each of which manages a subring of user nodes. To simplify the description, we use a vector numbering scheme to identify junction nodes and the user nodes that belong to their subrings. In Figure 2, the junction nodes are numbered 3.0 and 5.0.

As in the normal Beehive token ring, activity on the ring is conceived as a *train* that starts from the zero node, chugs one car per time slot through a sequence of nodes, and then ends up back at the zero node. However, the idea of the multiring extension is that usually the train will bypass the subrings, thus getting around the main ring with much less latency. Only when a user node on a subring has cars it wants to add to the train will the train take the local tour through the subring.

| user nodes | locks | runtime (sec) | trace length | distinct states | bug name | violation |
|---|---|---|---|---|---|---|
| 2 | 1 | 3 | 13 | 179 | BugOmitCheckReqP | InvLockMutex |
| 2 | 1 | 3 | $6 + 6^*$ | 263 | BugContinuousDoV | RequestCompletion |
| 2 | 1 | 3 | $16 + 7^*$ | 236 | BugOmitCheckDoV | LockForceReleasing |
| 2 | 1 | 3 | $11 + 10^*$ | 148 | BugOmitClaimLock | LockAcquisition |

Table 2: Model checking results for detecting intentional bugs in the ring lock protocol specification.

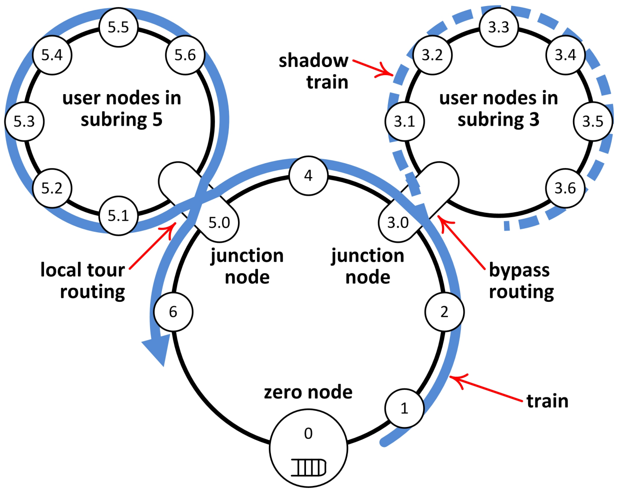

Figure 2: Multiring extension.

The reason for having bypass trains and local tour trains is to keep the multiring lock protocol as similar to the ring lock protocol as possible. Normally, trains bypass the subrings, thus achieving lower latency. However, when a user node on a subring wants to request a lock operation, that subring goes into local tour mode so that the user node can add its request car to the train and then inspect the result when its car returns on the next train. The tricky part in this design is that a junction node forwarding a bypass train has to be able to fail a *reqp* that would have been failed by one of its user nodes if the train had been sent on a local tour.

One could imagine other designs. The most obvious alternative design would be to arrange for subring nodes to somehow convey their new cars to the junction node, which would add them to the end of the train when it went by. There would be no need of local tour routing, but there is the problem of arranging for the junction node to buffer the new cars until they can be added to the train. We do not further investigate this alternative design.

In the bypass and local tour routing design, the routing of the train is controlled independently at each junction node. In the example shown in Figure 2, no user node on subring 3 wanted to add cars to the train, so junction node 3.0 sent the train past on bypass routing. However, on subring 5, some user node did want to add cars to the train, so junction node 5.0 sent the train on the local tour through subring 5.

There are many ways that could be used to inform a junction node that one of its subring user nodes wants to add cars to the train. For example, a simple wire-or signal wire could be used.

The routing decision of bypass or local tour must be made by the junction node no later than the time the train engine (the *token*) reaches the junction node on the main ring. Once the token has been sent in one direction, the entire train must follow.

When a junction node sends the train past in bypass routing, there may be cars on the train that one or more of its subring nodes need to hear about. For example, in order to force the release of a lock, whichever user node currently holds the lock needs to hear about it. For this reason, when the junction node sends the train past in bypass routing, it makes a copy of the train that it sends around its subring. We call this copy a *shadow train*. When the cars on a shadow train come back around the subring to the junction node, they are discarded.

The subring nodes can read the cars on a shadow train, which is of course the primary purpose of the shadow train. However, subring nodes are not permitted to rewrite cars on a shadow train nor are they permitted to add cars to a shadow train.

There are many techniques that can be used to distinguish a shadow train from a real train. The simplest technique, which we adopt, is just to drop the *token* from a shadow train. The cars on a shadow train should be imagined as being self-propelled.

## 5 The multiring lock protocol

The multiring extension supports a small number of global hardware locks that are managed using the *multiring lock protocol*. We describe the protocol, prove its safety and liveness, and discuss how we checked the protocol using a formal specification.

### 5.1 Description

The *multiring lock protocol* is based on the ring lock protocol and permits the same three actions to user nodes as described in Section 3.1. The main differences are in dealing with bypass trains.

The multiring lock protocol uses five special types of data cars: *reqp*, *failp*, *dov*, *gotp*, and *didv*. Each of these cars contains the node id of the node that creates the car and the lock id of a particular lock of interest. *reqp*, *failp*, and *dov* are used as in the ring lock protocol. When a subring user node wants to send a *reqp* or *dov* around the ring, it requests a real train from its junction node and it maintains this request until the real train circles the ring and comes back a second time. It adds the *reqp* or *dov* to the first train and rewrites it to *null* on the second train.

The tricky part of the multiring lock protocol is arranging for a junction node to fail a *reqp* on a bypass train that would have been failed by one of its subring user nodes if the train had been sent through its subring on a local tour. In order to do this, the junction node has to know which locks are held by its subring user nodes. The junction node keeps this information in a small array of subring lock bits. A subring user node uses the *gotp* and *didv* cars to update this array of bits in the junction node.

The *gotp* and *didv* cars are not just an optimization, but rather they convey information necessary for the junction node to be able to handle bypass trains properly.

Coordinating the actions of the junction node and its subring user nodes is essential. This is accomplished as follows.

The junction node only fails a bypass *reqp*. A *reqp* on a local tour train is left to be handled by the subring user nodes.

When a user node receives a successful *reqp* that has circled the ring, it claims the lock and rewrites the *reqp* to *gotp*. In the case of a subring user node, the *gotp* continues along the subring and eventually comes back out to the junction node, to inform it that the lock was claimed.

An unsuccessful *reqp* returns as a *failp* which the user node rewrites to *null* as in the ring lock protocol. The junction node does not need to be informed of failed attempts to acquire locks.

When a user node releases a lock as a permitted action, it requests a real train and adds a *didv*. In the case of a subring user node, the *didv* continues along the subring and eventually comes back out to the junction node, to inform it that the lock was released.

When a user node receives a *dov* that causes it to release a lock, it requests a real train an adds a *didv*, just as in the case of releasing a lock as a permitted action. The received *dov* can be on a real train or on a shadow train.

Between the time a user node releases a lock and adds the resulting *didv* to a train, the user node continues to fail *reqp*'s for that lock as if it still held the lock. Observe that a *didv* results from releasing a lock whereas a *gotp* results from claiming a lock. Continuing to fail *reqp*'s until the *didv* is added to the train guarantees that the *didv* will be on the train ahead of any *gotp* that results from a subsequent claim of the same lock.

The junction node's array of subring lock bits is updated by *gotp* and *didv* cars written by its subring user nodes, as these cars come out of the subring. The junction node can tell that the cars came from one of its subring user nodes by examining the node id recorded in the car. Hence the cars do not have to be nullified as they continue on through the ring.

The zero node discards any *didv* or *gotp* that it receives. Hence these cars do not circle the ring.

### 5.2 Safety and liveness

We now proceed to prove that the multiring lock protocol satisfies the mutual exclusion and lock acquisition properties. Our proof is based on the corresponding proofs in Section 3.2 for the ring lock protocol. First we need to adapt some of the definitions.

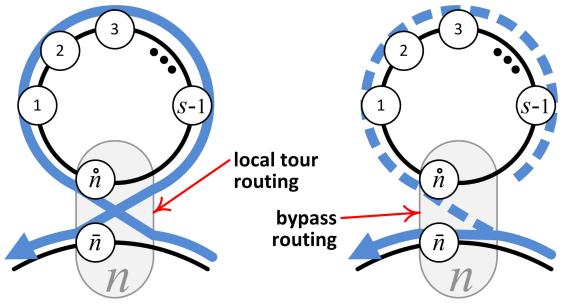

Figure 3: Internals of junction node $n$.

For the purpose of the proving safety and liveness, we pretend that a junction node $n$ is actually two nodes, $\bar{n}$ and $\mathring{n}$, where $\bar{n}$ buffers cars that are destined for the following node on the main ring and $\mathring{n}$ buffers cars that are destined for the local tour around the subring.

See Figure 3. When junction node $n$ is in local tour mode, cars arriving on the main ring are forwarded into $\mathring{n}$, then forwarded around the subring, then forwarded into $\bar{n}$, and finally forwarded into the next node on the main ring. When junction node $n$ is in bypass mode, cars arriving on the main ring are forwarded into $\bar{n}$, and then forwarded into the next node on the main ring.

We define the odometer for real cars only. We adapt the definition of $\varphi_{c,t}$, the *odometer* of car $c$ at time $t$, as follows. Instead of merely counting each forwarding step, the adapted odometer sums a weighted value as follows. For a forwarding step that represents being forwarded by a junction node in bypass mode (that is, from the previous main ring node directly into $\bar{n}$), the weighted value is $s + 1$, where $s$ is the number of nodes in the subring, counting the junction node. For all other forwarding steps, the weighted value is 1. The effect of being forwarded in bypass mode is as if the odometer counted passing through all of the nodes in the subring instantaneously.

Since a junction node routes all cars on a given train the same way, Property 1 and Property 2 of odometers in the ring lock protocol are also true of odometers in the multiring lock protocol. The proofs go through in exactly the same way.

Now we prove that the multiring lock protocol guarantees mutual exclusion. In the multiring lock protocol, user node $n_1$ claims a lock when its unchanged *reqp* car $c_1$ returns having circled the ring on a real train. Let $t_1$ be the time at which this happens. Observe that $\varphi_{c_1,t_1}$ must count once for every user node in the system, twice for every junction node, and once for the zero node. Let this value be $R$. We have $\varphi_{c_1,t_1} = R$.

By Property 1, no other node can have a car on the same train whose odometer at time $t_1$ is $R$ and therefore we do not have to worry about multiple nodes claiming the same lock at the same time. The remainder of the proof goes through exactly as for the ring lock protocol provided that whenever a subring user node holds a lock while its junction node is forwarding a train in bypass mode, the junction node indeed believes that one of its subring user nodes holds the lock. This fact is established as follows.

A subring user node can only claim a lock while it is receiving a real train. Hence at that time its junction node cannot be forwarding a train in bypass mode. Furthermore, at the time the subring user node claims the lock, it rewrites its *reqp* to *gotp* on the real train. During the time it takes this car to chug along the real train back out to the junction node, the junction node has to have remained in local tour mode.

So the only place a problem could arise is with the protocol for informing the junction node when the lock has been released. The only way to notify the junction node that one of its subring user nodes has released a lock is by adding a *didv* to a real train. Before this car has been added, the user node in question continues to act as though it still held the lock. So the sequence of *gotp* and *didv* cars for a given lock that appear on the same train record exactly the sequence in which the lock was claimed and released. Hence when the end of the local tour train emerges from the subring, the junction node will know for each lock the final state of whether or not any user node on its subring is acting as though it holds the lock.

This completes the proof that the multiring lock protocol guarantees mutual exclusion.

It is easy to establish that if a lock is desired infinitely often, eventually some node holds the lock. Assume that no node currently holds the lock. There are only two possible impediments to successfully acquiring the lock. (1) The last node to hold the lock is still acting as though it holds the lock, because it has not yet added the *didv* to the end of a real train. But in this case the node is requesting a real train, and eventually one will arrive and the *didv* will be added. (2) A junction node still believes that one of its subring nodes holds the lock. But in this case eventually a *didv* will arrive to correct this belief. So if a lock that is not held is always eventually desired, eventually there will be no impediments and the lock will be acquired.

It is important to observe that forcing the release of a lock is not quite as strong an action in the multiring lock protocol as in the ring lock protocol. In the ring lock protocol, when a user node receives back its *dov*, the lock has been released and, assuming no other node has already jumped in to request the lock, a subsequent request to acquire the lock will succeed. However, in the multiring lock protocol, when a user node receives back its *dov*, the lock might not yet be released and, furthermore, even after the lock is released, a subsequent (too prompt) attempt to acquire the lock might not succeed. This happens because of delay in informing the junction node that its subring user node has released a lock because of a *dov*. Note that the *dov* might have arrived on a shadow train, in which case the user node has to request a real train so that it can add a *didv* to inform the junction node that it released the lock.

## 5.3 Checking the protocol

Appendix B contains a formal specification of the multiring lock protocol, written in TLA+ [1]. The structure and much of the text is identical to the specification of the ring lock protocol discussed in Section 3.3. The main differences are (1) vector node numbering, (2) a few additional message types, (3) a few additional activites and changes in the behavior of the user node, and (4) lots of definitions related to the junction node.

The specification is parameterized by a *user node configuration*, which is a vector that lists the number of user nodes on the subring at each position around the main ring. A value of zero means that a user node appears on the main ring, instead of a junction node with a subring. The following examples should help explain this notation:

$\langle 2\rangle$ describes a main ring containing one subring that contains two user nodes.

$\langle 2, 1\rangle$ describes a main ring containing two subrings, the first of which contains two user nodes and the second of which contains one user node.

| **user node configuration** | **locks** | **runtime (sec)** | **depth** | **distinct states** |
|---|---|---|---|---|
| $\langle 1\rangle$ | 1 | 4 | 19 | 56 |
| $\langle 2\rangle$ | 1 | 15 | 26 | 623 |
| $\langle 0, 1\rangle$ | 1 | 17 | 26 | 813 |
| $\langle 0, 2\rangle$ | 1 | 162 | 31 | 6421 |
| $\langle 1, 0\rangle$ | 1 | 11 | 24 | 544 |
| $\langle 1, 1\rangle$ | 1 | 24 | 30 | 1004 |
| $\langle 1, 2\rangle$ | 1 | 277 | 35 | 7618 |
| $\langle 2, 0\rangle$ | 1 | 133 | 31 | 1280 |
| $\langle 2, 1\rangle$ | 1 | 359 | 35 | 10427 |
| $\langle 2, 2\rangle$ | 1 | 3689 | 40 | 69035 |
| $\langle 0, 0, 1\rangle$ | 1 | 170 | 31 | 7857 |
| $\langle 0, 0, 2\rangle$ | 1 | 1962 | 37 | 54644 |
| $\langle 0, 0, 0, 1\rangle$ | 1 | 1937 | 39 | 60375 |
| $\langle 0, 0, 0, 2\rangle$ | 1 | 20050 | 44 | 398605 |

Table 3: Model checking results for the multiring lock protocol specification. No violations were reported.

$\langle 2, 2\rangle$ describes a main ring containing two subrings, each of which contains two user nodes.

$\langle 0, 0, 2\rangle$ describes a main ring containing two user nodes followed by a subring that contains two user nodes. Each value of zero indicates a user node that appears on the main ring, instead of a junction node with a subring.

We used the TLC model checker to check several models of the ring lock protocol specification, with the results listed in Table 3. No violations of invariants or of temporal properties were reported. As is typical with model checking, the number of distinct states and the runtime of the model checker increase enormously as the model size increases, which makes checking feasible only for small models.

In order to gain confidence that model checking would actually find errors if the specification were buggy, it helps to intentionally introduce bugs and see if the model checker reports a violation. This is the purpose of the various *Bug...* definitions in the specification. A model configuration file can override these definitions to introduce the bug.

We ran the TLC model checker on such buggy configurations, described as follows:

| user node configuration | locks | runtime (sec) | trace length | distinct states | bug name | violation |
|---|---|---|---|---|---|---|
| $\langle 2\rangle$ | 1 | 8 | 16 | 269 | BugOmitCheckReqP | InvLockMutex |
| $\langle 1\rangle$ | 1 | 3 | $4+6^*$ | 48 | BugContinuousDoV | RequestCompletion |
| $\langle 1,1\rangle$ | 1 | 10 | $17+9^*$ | 1019 | BugOmitCheckDoV | LockForceReleasing |
| $\langle 1\rangle$ | 1 | 4 | $0+29^*$ | 33 | BugOmitClaimLock | LockAcquisition |
| $\langle 1,1\rangle$ | 1 | 9 | $17+9^*$ | 1009 | BugOmitCheckShadowDoV | LockForceReleasing |
| $\langle 1\rangle$ | 1 | 3 | 14 | 42 | BugOmitDidV | InvBypassSubhold |
| $\langle 0,2\rangle$ | 1 | 117 | 22 | 5039 | BugOmitDidvCheckReqP | InvBypassSubhold |

Table 4: Model checking results for detecting intentional bugs in the multiring lock protocol specification.

**BugOmitCheckReqP** A node holding a lock omits to fail a *reqp* for that lock, resulting in the error that the lock may be claimed by more than one node at the same time. (Same bug as ring lock protocol.)

**BugContinuousDoV** A node omits to process a *dov* it sent that circled the ring, resulting in the error that a request to force the release of a lock will never complete. (Same bug as ring lock protocol.)

**BugOmitCheckDoV** A node holding a lock omits to release the lock when it receives a *dov* for that lock, resulting in the error that a request to force the release of a lock may fail to release the lock. (Same bug as ring lock protocol.)

**BugOmitClaimLock** A node omits to claim a lock when its *reqp* returns after circling the ring, resulting in the error that a lock desired infinitely often may never be acquired. (Same bug as ring lock protocol.)

**BugOmitCheckShadowDoV** A node holding a lock omits to release the lock when it receives a *dov* for that lock on a shadow train, resulting in the error that a request to force the release of a lock may fail to release the lock.

**BugOmitDidV** A node omits to send a *didv* when software requests it to release a lock it holds, resulting in the error that a junction node not in local tour mode has incorrect information about what locks its subring nodes hold.

**BugOmitDidvCheckReqP** A node that has released a lock but not yet sent the corresponding *didv* omits to fail a *reqp* for that lock on a real train, resulting eventually in the error that a junction node not in local tour mode has incorrect information about what locks its subring nodes hold. This bug is discussed further in Section 5.4.

Violations were reported in every case even with very small model sizes. Table 4 shows the results. The *trace length* is the number of states in the counterexample produced by TLC. An infinite trace is required to exhibit a counterexample of a temporal property; this consits of an initial $i$ states followed by a cycle of $c$ states, which is listed as $i + c^*$.

Because the number of states explodes enormously as the model size increases, we looked for the smallest model that would exhibit the violation in each case.

## 5.4 An interesting bug

*BugOmitDidvCheckReqP* is an interesting bug. In this bug, a node that has released a lock but not yet sent the corresponding *didv* omits to fail a *reqp* for that lock on a real train. In a suitable configuration, this omission can result in a second user node on the subring claiming the lock and rewriting the *reqp* to a *gotp*, while later at the end of the train the first node finally adds its *didv*. As the train comes out of the subring, the junction node first sees the *gotp* and then sees the *didv*, both for the same lock.

If the junction node were counting the number of its subring nodes that claimed the lock, the count would go up to 2 when it saw the *gotp* and then back to 1 when it saw the *didv*. But we have a simpler design in which the junction node just records one bit per lock based on the last information it receives. So the result is that after

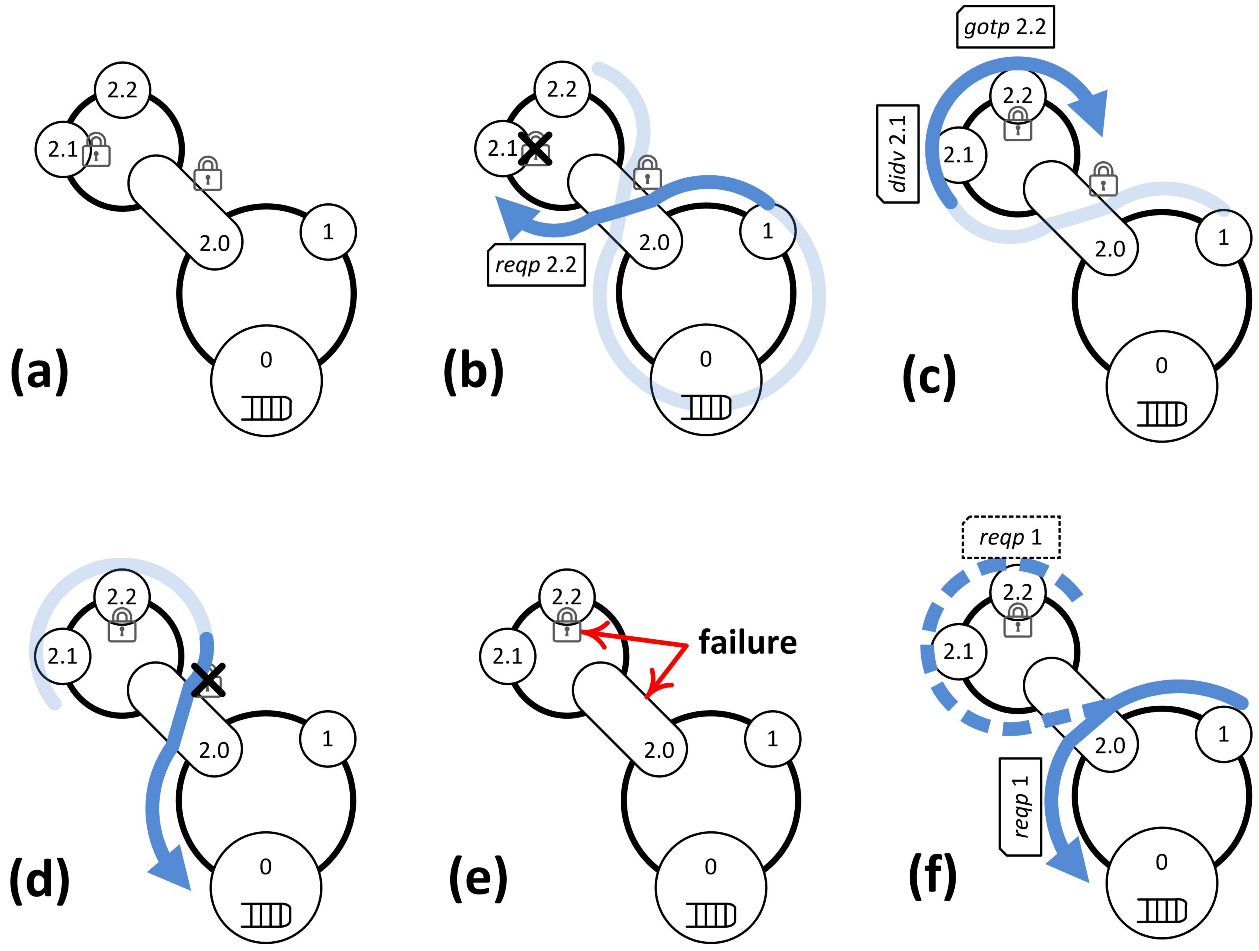

Figure 4: Counterexample for *BugOmitDidvCheckReqP*.

receiving the *didv*, the junction node ends up erronious believing that none of its subring nodes holds the lock.

Figure 4 shows a detailed counterexample. The configuration contains user node 1 on the main ring and two user nodes 2.1 and 2.2 on a subring. Node 1 is not necessary to the counterexample, but it shows why the invariant is important. First node 2.1 runs the multiring lock protocol and claims the lock (Figure 4(a)). Then node 2.2 decides it wants the lock and sends a *reqp*. The *reqp* circles the ring but before it arrives node 2.1 decides to release the lock (Figure 4(b)). Node 2.1 knows it needs to send a *didv* but, having the bug, it lets the *reqp* for node 2.2 pass by unchanged. Node 2.2 receives its successful *reqp*, rewrites it to *gotp*, and claims the lock. Meanwhile node 2.1 adds its *didv* to the end of the train (Figure 4(c)). As the train emerges from the subring, the last car junction node 2.0 sees is the *didv*, so it erroniously thinks that no node on its subring holds the lock (Figure 4(d)). Once the train is completely out, junction node 2.0 exits local tour mode, causing a violation of the invariant *InvBypassSubhold* (Figure 4(e)). This invariant is important, because if user node 1 subsequently decided it wanted the lock, its *reqp* could go past junction node 2.0 in bypass mode without failing (Figure 4(f)). This could result in both node 1 and node 2.2 holding the lock at the same time, thus violating mutual exclusion.

## 5.5 Hierarchical multiring

The multiring extension can be applied recursively to form subrings within subrings, creating a hierarchical multiring structure as illustrated in Figure 5. In this illustration, nodes 3.4, 5.4, and 5.6 have been replaced with junction nodes 3.4.0, 5.4.0, and 5.6.0 respectively, each of which manages a lower level subring.

In the hierarchical multiring, a junction node routes an arriving real train on a local tour if a real train is requested by any node within it subring hierarchy. If there is no such request, it sends the real train past as a bypass train and sends a shadow copy down its subring. An arriving shadow train is always just copied down its subring as a shadow train.

In the example in Figure 5, junction nodes 5.0 and 5.6.0 are sending the train on a local tour, while junction nodes 3.0 and 5.4.0 are sending the train on a bypass. Junction node 3.4.0 is making a shadow copy of a shadow train.

The multiring lock protocol applies without modification to the hierarchical multiring. Three features of the multiring lock protocol make this possible.

First, the multiring lock protocol never makes any change to a car on a shadow train. The cars on a shadow train are read only. Hence it does not matter where a shadow train goes, as long as its cars reach all user nodes within the subring, arriving in the order they appear on the train. The hierarchical multiring preserves this property.

Second, in the multiring lock protocol, a user node informs its junction node that it has claimed a lock by changing its returned *reqp* to *gotp*. The junction node learns about this when the *gotp* exits the subring, by examining the car's node id and noticing that it belongs to a node within its subring. However, the junction node makes no change to the *gotp*, leaving it on the train for any subsequent junction node to see, such as, for example, a superior junction node. Of course, in the basic multiring structure there are no superior junction nodes, but this works perfectly in the hierarchical multiring.

Third, in the multiring lock protocol, a user node informs its junction node that it has released a lock by adding a *didv* to a real train. This works in the hierarchical multiring exactly the same as the *gotp* just discussed.

So the hierarchical multiring structure can use the multiring lock protocol without change.

Using the hierarchical multiring structure enables decreasing the latency of sending a car around a ring of $n$ nodes from $O(n)$ for a basic token ring or multiring to $O(\log n)$. The theoretical improvement can begin to be realized even in reasonably small sizes. For example, consider a system with 64 user nodes. Let us count the number time slots between when a user node adds a data car to an emty train until that car returns, assuming no other demand on the system. The Beehive token ring (described in Section 2) requires 66 time slots. A multiring structure of 8 rings of 8 user nodes requires 19 time slots: 10 to circle the main ring and 9 to circle the subring. A hierarchical multiring structure of 4 rings of 4 subrings of 4 user nodes requires 16 time slots: 6 to circle the main ring, 5 to circle the subring, and 5 to circle the subsubring.

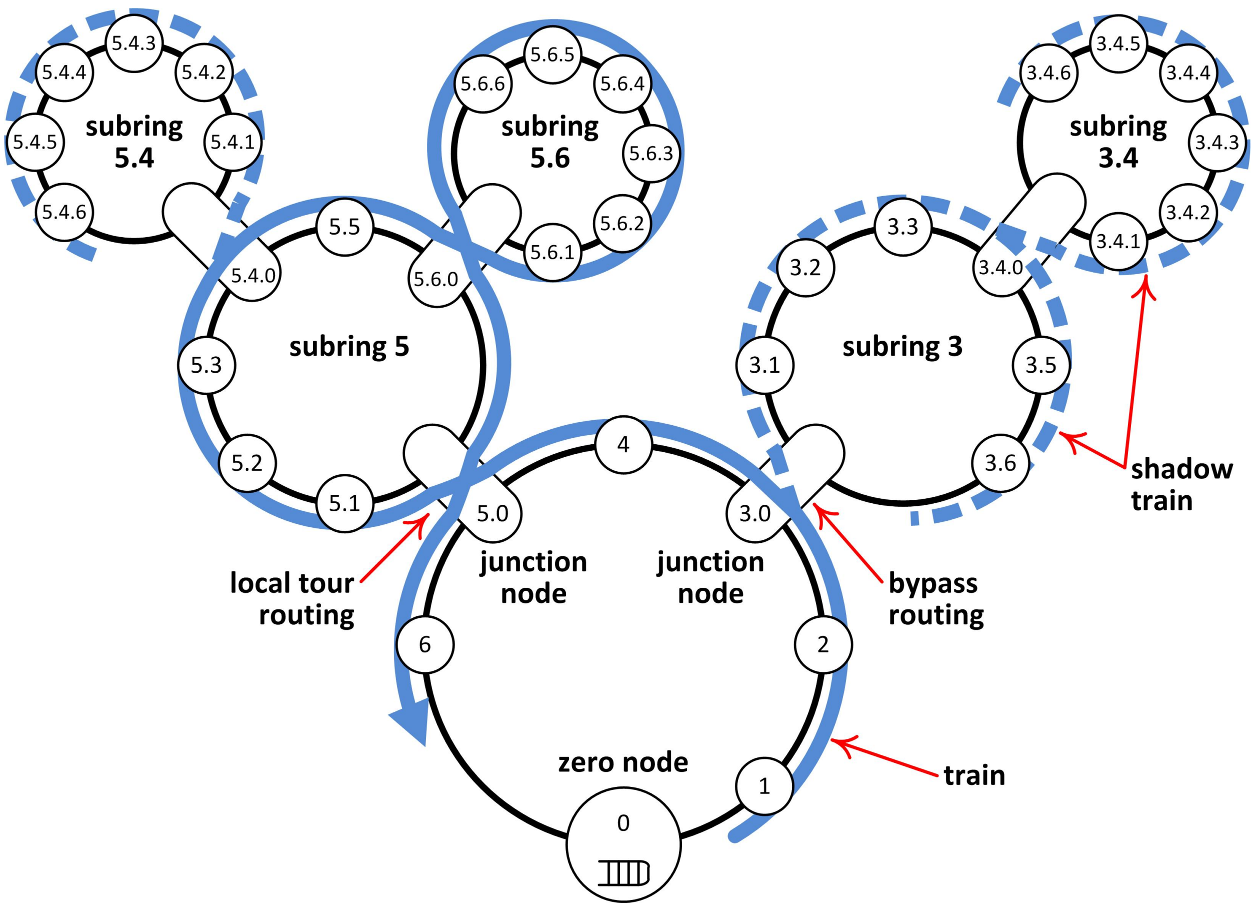

Figure 5: Hierarchical multiring.

## 6 Conclusion

The Beehive ring lock protocol was designed to exploit the Beehive token ring by performing lock acquisition with one cycle of the ring and lock release with no ring traffic at all. However, the justification of why the protocol is correct is fairly subtle. In such a case, a formal specification is useful for eliminating disagreements over details of the protocol.

The multiring extension adds considerable complexity to the lock protocol and introduces the possibility for many exciting bugs. One interesting result of adding hierarchy to the ring structure was the loss of the original overhead-free lock release. In the multiring extension, the superior must be informed of lock release so that it can act properly on behalf of its inferior nodes. Given this added complexity, using a formal specification is even more necessary for eliminating disagreements and checking correctness.

## Acknowledgements

Chuck Thacker designed and implemented the Beehive ring lock protocol, along with most of the rest of the Beehive system, and patiently answered many questions about the exact operation of Beehive. Lintao Zhang originally proposed the multiring extension. John Davis provided many helpful comments on this paper.

## References

[1] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[2] T. L. Rodeheffer. Code generation for the Beehive ISA. Technical Report MSR-TR-2010-113, Microsoft Research, Aug. 2010.

[3] C. Thacker. Beehive: A many-core computer for FPGAs, Jan. 2010. Unpublished.

## A Ring lock protocol specification

MODULE $RingLock$

EXTENDS $Naturals$, $Sequences$, $FiniteSets$, $TLC$

Specification of the Beehive ring lock protocol.

VARIABLE $state$

CONSTANT $Lock$ set of lock identifiers
CONSTANT $NumNode$ number of user nodes

Various bugs (for testing the model checking).

$BugOmitClaimLock \triangleq$ FALSE omit to claim an acquired lock
$BugOmitCheckReqP \triangleq$ FALSE omit to fail a *reqp* for held lock
$BugOmitCheckDoV \triangleq$ FALSE omit to release a lock on a *dov*
$BugContinuousDoV \triangleq$ FALSE forget to process returned *dov*

---

Node identifier.

A node is identified by a number. Node 0 is the zero node.

$Node \triangleq 0 \,..\, NumNode$ node identifier

$ZeroNode \triangleq \{n \in Node : n = 0\}$ set of zero nodes
$UserNode \triangleq \{n \in Node : n \neq 0\}$ set of user nodes

---

Type and initial value of an array of held locks.

$Hold \triangleq [Lock \rightarrow \text{BOOLEAN}]$ map from lock *id* to TRUE / FALSE

$InitHold \triangleq [lock \in Lock \mapsto \text{FALSE}]$ no locks held

---

Useful TLA+ definitions.

The sum of $f[x]$ for all $x$ in DOMAIN $f$.

```
$Sum(f) \triangleq$
  LET
    $DSum[S \in$ SUBSET DOMAIN $f] \triangleq$
      LET
        $x \triangleq$ CHOOSE $e \in S$ : TRUE
      IN
        IF $S = \{\}$ THEN 0 ELSE $f[x] + DSum[S \setminus \{x\}]$
  IN
    $DSum[$DOMAIN $f]$
```

---

MESSAGES

Define the structure of each type of message.

All messages have a type. Messages specific to the lock protocol also have a lock and a source.

|  |  |
|---|---|
| type | a string unique to this type of message. |
| lock | the lock the message is about. |
| source | the node that originally created the message. |

```
$MsgToken \triangleq [type : \{\text{“token”}\}]$
$MsgNull \triangleq [type : \{\text{“null”}\}]$
$MsgIdle \triangleq [type : \{\text{“idle”}\}]$
$MsgReqP \triangleq [type : \{\text{“reqp”}\}, lock : Lock, source : Node]$
$MsgFailP \triangleq [type : \{\text{“failp”}\}, lock : Lock, source : Node]$
$MsgDoV \triangleq [type : \{\text{“dov”}\}, lock : Lock, source : Node]$

$Msg \triangleq \{\}$
  $\cup MsgToken$
  $\cup MsgNull$
  $\cup MsgIdle$
  $\cup MsgReqP$
  $\cup MsgFailP$
  $\cup MsgDoV$
```

Convenience operators to construct messages.

```
$MkMsgToken \triangleq$ CHOOSE $m \in MsgToken$ : TRUE
$MkMsgNull \triangleq$ CHOOSE $m \in MsgNull$ : TRUE
$MkMsgIdle \triangleq$ CHOOSE $m \in MsgIdle$ : TRUE
$MkMsgReqP(l, s) \triangleq$ CHOOSE $m \in MsgReqP : m.lock = l \land m.source = s$
$MkMsgFailP(l, s) \triangleq$ CHOOSE $m \in MsgFailP : m.lock = l \land m.source = s$
$MkMsgDoV(l, s) \triangleq$ CHOOSE $m \in MsgDoV : m.lock = l \land m.source = s$
```

NODE ACTIVITIES

Define the structure of each type of activity.

All activities have a type. Activities specific to the lock protocol also have a lock.

| | |
|---|---|
| type | a string unique to this type of activity. |
| lock | the lock the activity is about. |

$ActIdle \triangleq [type : \{\text{"idle"}\}]$
$ActSendReqP \triangleq [type : \{\text{"send\_reqp"}\}, lock : Lock]$
$ActSendDoV \triangleq [type : \{\text{"send\_dov"}\}, lock : Lock]$
$ActWaitReqP \triangleq [type : \{\text{"wait\_reqp"}\}, lock : Lock]$
$ActWaitDoV \triangleq [type : \{\text{"wait\_dov"}\}, lock : Lock]$

$Act \triangleq \{\}$
  $\cup\ ActIdle$ idle
  $\cup\ ActSendReqP$ want to send a *reqp*
  $\cup\ ActSendDoV$ want to send a *dov*
  $\cup\ ActWaitReqP$ waiting for *reqp* to come back
  $\cup\ ActWaitDoV$ waiting for *dov* to come back

Convenience operators to construct activities.

$MkActIdle \triangleq \text{CHOOSE } a \in ActIdle : \text{TRUE}$
$MkActSendReqP(l) \triangleq \text{CHOOSE } a \in ActSendReqP : a.lock = l$
$MkActSendDoV(l) \triangleq \text{CHOOSE } a \in ActSendDoV : a.lock = l$
$MkActWaitReqP(l) \triangleq \text{CHOOSE } a \in ActWaitReqP : a.lock = l$
$MkActWaitDoV(l) \triangleq \text{CHOOSE } a \in ActWaitDoV : a.lock = l$

NODE STATE

Node state for a zero node.

*buf* buffers cars from one train to the next. Everything from the head of *buf* up to but not including a *token* car is part of the current train. Everything from a *token* car on back is part of the next train.

$ZeroNodeState \triangleq$ type definition
[
  $id : ZeroNode,$ the *id* of this node
  $out : Msg,$ sending on ring
  $buf : Seq(Msg)$ buffer cars from one train to the next

```
]

$InitZeroNodeState(n) \triangleq$    initial value
[
  $id \mapsto n,$
  $out \mapsto MkMsgToken,$  initially sending a token $msg$
  $buf \mapsto \langle\rangle$  empty buffer
]

$MsgsInZeroNodeState(ns) \triangleq$    set of all messages
  $\{ns.out\} \cup \{ns.buf[x] : x \in 1 \,..\, Len(ns.buf)\}$

$CountTokensInZeroNodeState(ns) \triangleq$    number of tokens stored
  LET
    $Cnt(m) \triangleq$ IF $m \in MsgToken$ THEN $1$ ELSE $0$
  IN
    $Cnt(ns.out) + Sum([x \in 1 \,..\, Len(ns.buf) \mapsto Cnt(ns.buf[x])])$
```

Node state for a user node.

```
$UserNodeState \triangleq$    type definition
[
  $id$      : $UserNode,$    the $id$ of this node
  $out$     : $Msg,$    sending on ring
  $intrain$   : BOOLEAN ,    received token and not yet idle
  $act$     : $Act,$  current activity of this node
  $hold$     : $Hold$  map of locks held by this node
]

$InitUserNodeState(n) \triangleq$    initial value
[
  $id$     $\mapsto n,$
  $out$     $\mapsto MkMsgIdle,$    sending idle $msg$
  $intrain$    $\mapsto$ FALSE,    not in a train
  $act$    $\mapsto MkActIdle,$    currently idle
  $hold$     $\mapsto InitHold$    no locks held
]

$MsgsInUserNodeState(ns) \triangleq$    set of all messages
  $\{ns.out\}$
```

```
$CountTokensInUserNodeState(ns) \triangleq$    number of tokens stored
  LET
    $Cnt(m) \triangleq$ IF $m \in MsgToken$ THEN $1$ ELSE $0$
  IN
    $Cnt(ns.out)$
```

General node state.

```
$NodeState \triangleq \{\}$  type definition
   $\cup\ ZeroNodeState$
   $\cup\ UserNodeState$

$InitNodeState(n) \triangleq$    initial value
  IF $n \in ZeroNode$ THEN $InitZeroNodeState(n)$ ELSE
  IF $n \in UserNode$ THEN $InitUserNodeState(n)$ ELSE
  $Assert($FALSE, "impossible"$)$

$MsgsInNodeState(ns) \triangleq$    set of all messages
  IF $ns.id \in ZeroNode$ THEN $MsgsInZeroNodeState(ns)$ ELSE
  IF $ns.id \in UserNode$ THEN $MsgsInUserNodeState(ns)$ ELSE
  $Assert($FALSE, "impossible"$)$

$CountTokensInNodeState(ns) \triangleq$    number of tokens stored
  IF $ns.id \in ZeroNode$ THEN $CountTokensInZeroNodeState(ns)$ ELSE
  IF $ns.id \in UserNode$ THEN $CountTokensInUserNodeState(ns)$ ELSE
  $Assert($FALSE, "impossible"$)$
```

Given the current *intrain* value $i$, compute the new *intrain* value after the arrival of message $m$.

```
$UpdateIntrain(i, m) \triangleq$
  IF $m \in MsgToken$ THEN TRUE ELSE    start of train
  IF $m \in MsgIdle$ THEN FALSE ELSE    end of train
  $i$
```

The set of messages the zero node should discard, rather than recycle.

```
$ZeroNodeDiscardMsgs \triangleq \{\}$
  $\cup\ MsgIdle$
  $\cup\ MsgNull$
```

On a zero node, append an arriving message $m$ to the buffer $buf$ if $m$ should be recycled.

```
$ZeroNodeAppendBuf(buf, m) \triangleq$
  IF $m \in ZeroNodeDiscardMsgs$ THEN $buf$ ELSE $Append(buf, m)$
```

Compute the new node state resulting from receiving the indicated message on the zero node.

A train consists of consecutive slots starting with a token and ending with an idle.

The zero node is in charge of starting the train. It recycles cars from the previous train in $buf$ . When $buf$ is nonempty, the head of $buf$ is a $token$ and the arriving message is an $idle$ , then it is possible to start a new train.

```
$ZeroNodeRecvMsg(ns, msg) \triangleq$
  IF $ns.id \notin ZeroNode$ THEN $Assert$(FALSE, "not a zero node")
   ELSE
  IF
      $\wedge\ ns.buf \neq \langle\rangle$
      $\wedge\ Head(ns.buf) \in MsgToken$
      $\wedge\ msg \in MsgIdle$
   THEN
    Start a new train.
    $[ns$ EXCEPT
    $!.out = Head(ns.buf),$
    $!.buf = ZeroNodeAppendBuf(Tail(ns.buf), msg)$
    $]$
   ELSE
  IF
      $\wedge\ ns.buf \neq \langle\rangle$
      $\wedge\ Head(ns.buf) \notin MsgToken$
   THEN
    Continue sending the current train.
    $[ns$ EXCEPT
    $!.out = Head(ns.buf),$
    $!.buf = ZeroNodeAppendBuf(Tail(ns.buf), msg)$
    $]$
   ELSE
    Send an $idle$ between trains.
    $[ns$ EXCEPT
    $!.out = MkMsgIdle,$
    $!.buf = ZeroNodeAppendBuf(ns.buf, msg)$
    $]$
```

Compute the new node state resulting from receiving the indicated message on a user node.

```
$UserNodeRecvMsg(ns, msg) \triangleq$
  IF $ns.id \notin UserNode$ THEN $Assert$(FALSE, "not a user node")
   ELSE
  IF
      $\land msg \in MsgIdle$
      $\land ns.act \in ActSendReqP$
      $\land ns.intrain$
   THEN
    Here is the $idle$ at the end of the train that we need to replace with our $reqp$ message. This extends the train
    by one car so we stay in the train.
    $[ns$ EXCEPT
    $!.out = MkMsgReqP(ns.act.lock, ns.id)$,   send the $reqp$
    $!.act = MkActWaitReqP(ns.act.lock)$,   wait for its return
    $!.intrain =$ TRUE
    $]$
   ELSE
  IF
      $\land msg \in MsgIdle$
      $\land ns.act \in ActSendDoV$
      $\land ns.intrain$
   THEN
    Here is the $idle$ at the end of the train that we need to replace with our $dov$ message. This extends the train
    by one car so we stay in the train.
    $[ns$ EXCEPT
    $!.out = MkMsgDoV(ns.act.lock, ns.id)$,   send the $dov$
    $!.act = MkActWaitDoV(ns.act.lock)$,   wait for its return
    $!.intrain =$ TRUE
    $]$
   ELSE
  IF
      $\land msg \in MsgReqP$
      $\land msg.source = ns.id$

      $\land Assert(ns.act = MkActWaitReqP(msg.lock)$, "return reqp unexpected")
      $\land Assert(ns.intrain$, "return reqp not in train")
     $\land$ $Assert(\neg ns.hold[msg.lock]$, “return reqp but lock held”)
   THEN
    Our *reqp* message returns. This can only happen when we are waiting for it, hence the assert.
    We own the lock. Nullify the *reqp* message.
    $[ns$ EXCEPT
    $!.out = MkMsgNull,$
    $!.act = MkActIdle,$
    $!.hold =$ IF $BugOmitClaimLock$ THEN $@$ ELSE
    $[@$ EXCEPT $![msg.lock] =$ TRUE$],$
    $!.intrain = UpdateIntrain(@, msg)$
    $]$
   ELSE
  IF
     $\land$ $msg \in MsgFailP$
     $\land$ $msg.source = ns.id$

     $\land$ $Assert(ns.act = MkActWaitReqP(msg.lock)$, “return failp unexpected”)
     $\land$ $Assert(ns.intrain$, “return failp not in train”)
     $\land$ $Assert(\neg ns.hold[msg.lock]$, “return failp but lock held”)
   THEN
    Our *reqp* message came back failed. This can only happen when we are waiting for it, hence the assert.
    We failed to get the lock. Nullify the *failp* message.
    $[ns$ EXCEPT
    $!.out = MkMsgNull,$
    $!.act = MkActIdle,$
    $!.intrain = UpdateIntrain(@, msg)$
    $]$
   ELSE
  IF
     $\land$ $msg \in MsgReqP$
     $\land$ $msg.source \neq ns.id$
     $\land$ $ns.hold[msg.lock]$

     $\land$ $\neg BugOmitCheckReqP$
   THEN
    Some one else’s *reqp* message for a lock we hold. Fail it.
    $[ns$ EXCEPT
    $!.out = MkMsgFailP(msg.lock, msg.source),$    note: keep source
    $!.intrain = UpdateIntrain(@, msg)$
    $]$
   ELSE
  IF
     $\land$ $msg \in MsgDoV$
     $\land$ $msg.source = ns.id$
     $\land$ $Assert(ns.act = MkActWaitDoV(msg.lock)$, "return dov unexpected")
     $\land$ $Assert(ns.intrain$, "return dov not in train")
     $\land$ $Assert(\neg ns.hold[msg.lock]$, "return dov but lock held")

     $\land$ $\neg BugContinuousDoV$
   THEN
    Our $dov$ message returns. This can only happen when we are waiting for the $dov$ to return.

    Nullify the message.
    $[ns$ EXCEPT
    $!.out = MkMsgNull$,
    $!.act = MkActIdle$,
    $!.intrain = UpdateIntrain(@, msg)$
    $]$
   ELSE
  IF
     $\land$ $msg \in MsgDoV$
     $\land$ $msg.source \neq ns.id$
     $\land$ $ns.hold[msg.lock]$

     $\land$ $\neg BugOmitCheckDoV$
   THEN
    Some one else's $dov$ message for a lock we hold. Clear our lock.
    $[ns$ EXCEPT
    $!.out = msg$,
    $!.hold = [@$ EXCEPT $![msg.lock] =$ FALSE$]$,
    $!.intrain = UpdateIntrain(@, msg)$
    $]$
   ELSE
    Pass the message unchanged.
    $[ns$ EXCEPT
    $!.out = msg$,
    $!.intrain = UpdateIntrain(@, msg)$
    $]$
```

---

STATE

```
$State \triangleq [$
  $nodes : [Node \rightarrow NodeState]$
$]$

$InitState \triangleq [$
```

```
  $nodes \mapsto [n \in Node \mapsto InitNodeState(n)]$
]
```

NEXT STATE RELATIONS

Compute the node from which node $n0$ takes its message.

```
$FromNode(n0) \triangleq$
  CHOOSE $n \in Node : (n + 1)\%(NumNode + 1) = n0$
```

Advance by one step.

Each node receives the message sent by the node earlier in the ring, and updates its state, computing the message it will send in the next step.

```
$NextStep \triangleq$
  LET
    $NN \triangleq state.nodes$

    How to update a zero node.
    $zeroupd(n) \triangleq$
      LET
        $ns \triangleq NN[n]$
        $msg \triangleq NN[FromNode(n)].out$
      IN
        $ZeroNodeRecvMsg(ns, msg)$

    How to update a user node.
    $userupd(n) \triangleq$
      LET
        $ns \triangleq NN[n]$
        $msg \triangleq NN[FromNode(n)].out$
      IN
        $UserNodeRecvMsg(ns, msg)$
  IN
    $state' =$
    $[state$ EXCEPT
    $!.nodes =$
    $[n \in Node \mapsto$
    IF $n \in ZeroNode$ THEN $zeroupd(n)$ ELSE
    IF $n \in UserNode$ THEN $userupd(n)$ ELSE
    $Assert($FALSE, “uncovered”$)$
    $]$
    $]$
```

Some user node decides to try to take a lock it does not hold.

```
$NextTakeUnheldLock \triangleq$
  $\exists\, node \in UserNode :$
  $\exists\, lock \in Lock :$
  LET
    $ns \triangleq state.nodes[node]$
  IN
     $\land\, ns.act \in ActIdle$
     $\land\, \neg ns.hold[lock]$
     $\land\, state' =$
    $[state$ EXCEPT
    $!.nodes[node].act = MkActSendReqP(lock)$
    $]$
```

Some user node decides to release a lock it holds.

```
$NextReleaseHeldLock \triangleq$
  $\exists\, node \in UserNode :$
  $\exists\, lock \in Lock :$
  LET
    $ns \triangleq state.nodes[node]$
  IN
     $\land\, ns.act \in ActIdle$
     $\land\, ns.hold[lock]$
     $\land\, state' =$
    $[state$ EXCEPT
    $!.nodes[node].hold[lock] =$ FALSE
    $]$
```

Some user node decides to release a lock it does not hold.

```
$NextReleaseUnheldLock \triangleq$
  $\exists\, node \in UserNode :$
  $\exists\, lock \in Lock :$
  LET
```

```
    $ns \triangleq state.nodes[node]$
  IN
    $\land ns.act \in ActIdle$
    $\land \neg ns.hold[lock]$
    $\land state' =$
    $[state$ EXCEPT
    $!.nodes[node].act = MkActSendDoV(lock)$
    $]$
```

---

INVARIANTS

The state must always be of the proper type.

```
$InvType \triangleq$
  $\land state \in State$
```

There is always exactly one token.

```
$InvUniqueToken \triangleq$
  LET
    Total count of tokens stored over all nodes.
    $total \triangleq Sum([n \in Node \mapsto CountTokensInNodeState(state.nodes[n])])$
  IN
    $total = 1$
```

No two user nodes can hold the same lock.

```
$InvLockMutex \triangleq$
  $\forall n1, n2 \in UserNode :$
  $\forall lock \in Lock :$
  $($
    $\land state.nodes[n1].hold[lock]$
    $\land state.nodes[n2].hold[lock]$
  $) \Rightarrow n1 = n2$
```

---

TEMPORAL ASSUMPTIONS

Liveness assumption.

We have to make explicit the temporal assumption that always eventually a time step happens on the ring. (Otherwise, the model permits infinite suttering.) Since the step is always enabled, weak fairness is sufficient.

$$
\begin{array}{l}
Liveness \triangleq \\
\quad \land\ \mathrm{WF}_{state}(NextStep)
\end{array}
$$

---

TEMPORAL PROPERTIES

For all user nodes, the node always eventually gets back to the idle state.

$RequestCompletion \triangleq$

$\quad \forall\, n \in UserNode :$

$\quad$ LET

$\quad\quad$ This user node is idle.

$\quad\quad idle \triangleq state.nodes[n].act \in ActIdle$

$\quad$ IN

$\quad\quad \Box\Diamond idle$

For all locks, some node desiring the lock leads to some node holding it.

$LockAcquisition \triangleq$

$\quad \forall\, lock \in Lock :$

$\quad$ LET

$\quad\quad$ Some node desires this lock.

$\quad\quad desired \triangleq \exists\, n \in UserNode :$

$\quad\quad state.nodes[n].act = MkActSendReqP(lock)$

$\quad\quad$ Some node holds this lock.

$\quad\quad held \triangleq \exists\, n \in UserNode :$

$\quad\quad\quad state.nodes[n].hold[lock]$

$\quad$ IN

$\quad\quad desired \leadsto held$

For all locks, some node wanting to $dov$ the lock leads to no node holding it.

```
$LockForceReleasing \triangleq$
  $\forall\, lock \in Lock :$
  LET
    Some node wants to $dov$ this lock.
    $wantdov \triangleq \exists\, n \in UserNode :$
    $state.nodes[n].act = MkActSendDoV(lock)$

    No node holds this lock.
    $free \triangleq \forall\, n \in UserNode :$
       $\neg state.nodes[n].hold[lock]$
  IN
    $wantdov \leadsto free$
```

FINAL SPECIFICATION

Initial state.

```
$Init \triangleq$
  $\land\, state = InitState$
```

Next state relation.

```
$Next \triangleq$
  $\lor\, NextStep$
  $\lor\, NextTakeUnheldLock$
  $\lor\, NextReleaseHeldLock$
  $\lor\, NextReleaseUnheldLock$
```

Specification.

```
$Spec \triangleq$
  $\land\, Init$
  $\land\, \Box[Next]_{state}$
  $\land\, Liveness$
```

THEOREM $Spec \Rightarrow$

$\land \Box InvType$
$\land \Box InvUniqueToken$
$\land \Box InvLockMutex$
$\land RequestCompletion$
$\land LockAcquisition$
$\land LockForceReleasing$

## B Multiring lock protocol specification

MODULE $MultiringLock$

EXTENDS $Naturals, Sequences, FiniteSets, TLC$

Specification of multiring lock protocol.

VARIABLE $state$

CONSTANT $Lock$ lock *id*
CONSTANT $NumNode$ map $1 \,..\, r \mapsto$ user nodes in subring $r$
0 means user node on the main ring

Various bugs (for testing the model checking).

$BugOmitClaimLock \triangleq$ FALSE omit to claim an acquired lock
$BugOmitCheckReqP \triangleq$ FALSE omit to fail a *reqp* for held lock
$BugOmitCheckDoV \triangleq$ FALSE omit to release a lock on a *dov*
$BugContinuousDoV \triangleq$ FALSE forget to process returned *dov*

$BugOmitCheckShadowDoV \triangleq$ FALSE omit to watch for *dov* on shadow train
$BugOmitDidV \triangleq$ FALSE omit sending a *didv*
$BugOmitDidvCheckReqP \triangleq$ FALSE omit to fail a *reqp* for a $didv[lock]$

Node identifier.

A node is identified by a sequence $v$ of numbers.

For the zero node the sequence is of length 1 and $v[1] = 0$.

For a user node on the main ring, the sequence is of length 1 and $v[1]$ is the position of the user node on the ring. $v[1] \neq 0$ because that is the zero node.

For a junction node that manages subring $r$, the sequence is of length 2 and $v[1] = r$ and $v[2] = 0$. The junction node is at position $r$ on the main ring. This must not conflict with the position of the zero node nor with the position of any user node on the main ring.

For a user node on subring $r$, the sequence is of length 2 and $v[1] = r$ and $v[2]$ is the position of the user node on subring $r$. $v[2] \neq 0$ because that is the position of the junction node that manages subring $r$.

Positions on the main ring where there is a user node.

$MainRingPosUserNode \triangleq \{r \in \text{DOMAIN } NumNode : NumNode[r] = 0\}$

Positions on the main ring where there is a junction node.

$MainRingPosJuncNode \triangleq \{r \in \text{DOMAIN } NumNode : NumNode[r] > 0\}$

The set of all zero nodes.

```
$ZeroNode \triangleq \{\langle 0 \rangle\}$
```

The set of all user nodes. A user node can be on the main ring or it can be on a subring.

```
$UserNode \triangleq$ LET
  $onmain \triangleq \{\langle r \rangle : r \in MainRingPosUserNode\}$
  $onring(r) \triangleq \{\langle r, i \rangle : i \in 1 \,..\, NumNode[r]\}$
IN
  $onmain \cup$ UNION $\{onring(r) : r \in MainRingPosJuncNode\}$
```

The set of all junction nodes.

```
$JuncNode \triangleq \{\langle r, 0 \rangle : r \in MainRingPosJuncNode\}$
```

The set of all nodes.

```
$Node \triangleq \{\}$
  $\cup\ ZeroNode$
  $\cup\ UserNode$
  $\cup\ JuncNode$
```

---

Useful TLA+ definitions.

The sum of $f[x]$ for all $x$ in DOMAIN $f$.

```
$Sum(f) \triangleq$
  LET
    $DSum[S \in$ SUBSET DOMAIN $f] \triangleq$
      LET
        $x \triangleq$ CHOOSE $e \in S$ : TRUE
      IN
        IF $S = \{\}$ THEN $0$ ELSE $f[x] + DSum[S \setminus \{x\}]$
  IN
    $DSum[$DOMAIN $f]$
```

---

MESSAGES

Define the structure of each type of message.

All messages have a type. Messages specific to the lock protocol also have a lock and a source.

| | |
|---|---|
| type | a string unique to this type of message. |
| lock | the lock the message is about. |
| source | the node that originally created the message. |

$MsgToken \triangleq [type : \{\text{"token"}\}]$
$MsgNull \triangleq [type : \{\text{"null"}\}]$
$MsgIdle \triangleq [type : \{\text{"idle"}\}]$
$MsgReqP \triangleq [type : \{\text{"reqp"}\}, lock : Lock, source : Node]$
$MsgFailP \triangleq [type : \{\text{"failp"}\}, lock : Lock, source : Node]$
$MsgDoV \triangleq [type : \{\text{"dov"}\}, lock : Lock, source : Node]$
$MsgGotP \triangleq [type : \{\text{"gotp"}\}, lock : Lock, source : Node]$
$MsgDidV \triangleq [type : \{\text{"didv"}\}, lock : Lock, source : Node]$

$Msg \triangleq \{\}$
$\quad \cup\ MsgToken$
$\quad \cup\ MsgNull$
$\quad \cup\ MsgIdle$
$\quad \cup\ MsgReqP$
$\quad \cup\ MsgFailP$
$\quad \cup\ MsgDoV$
$\quad \cup\ MsgGotP$
$\quad \cup\ MsgDidV$

Convenience operators to construct messages.

$MkMsgToken \triangleq \text{CHOOSE } m \in MsgToken : \text{TRUE}$
$MkMsgNull \triangleq \text{CHOOSE } m \in MsgNull : \text{TRUE}$
$MkMsgIdle \triangleq \text{CHOOSE } m \in MsgIdle : \text{TRUE}$
$MkMsgReqP(l, s) \triangleq \text{CHOOSE } m \in MsgReqP : m.lock = l \land m.source = s$
$MkMsgFailP(l, s) \triangleq \text{CHOOSE } m \in MsgFailP : m.lock = l \land m.source = s$
$MkMsgDoV(l, s) \triangleq \text{CHOOSE } m \in MsgDoV : m.lock = l \land m.source = s$
$MkMsgGotP(l, s) \triangleq \text{CHOOSE } m \in MsgGotP : m.lock = l \land m.source = s$
$MkMsgDidV(l, s) \triangleq \text{CHOOSE } m \in MsgDidV : m.lock = l \land m.source = s$

---

NODE ACTIVITIES

Define the structure of each type of activity.

All activities have a type. Activities specific to the lock protocol also have a lock.

| | |
|---|---|
| type | a string unique to this type of activity. |
| lock | the lock the activity is about. |

$ActIdle \triangleq [type : \{\text{"idle"}\}]$

$ActSendReqP \triangleq [type : \{\text{"send\_reqp"}\}, lock : Lock]$
$ActSendDoV \triangleq [type : \{\text{"send\_dov"}\}, lock : Lock]$
$ActWaitReqP \triangleq [type : \{\text{"wait\_reqp"}\}, lock : Lock]$
$ActWaitDoV \triangleq [type : \{\text{"wait\_dov"}\}, lock : Lock]$

$Act \triangleq \{\}$
  $\cup\ ActIdle$ idle
  $\cup\ ActSendReqP$ want to send a *reqp*
  $\cup\ ActSendDoV$ want to send a *dov*
  $\cup\ ActWaitReqP$ waiting for *reqp* to come back
  $\cup\ ActWaitDoV$ waiting for *dov* to come back

Convenience operators to construct activities.

$MkActIdle \triangleq \text{CHOOSE } a \in ActIdle : \text{TRUE}$
$MkActSendReqP(l) \triangleq \text{CHOOSE } a \in ActSendReqP : a.lock = l$
$MkActSendDoV(l) \triangleq \text{CHOOSE } a \in ActSendDoV : a.lock = l$
$MkActWaitReqP(l) \triangleq \text{CHOOSE } a \in ActWaitReqP : a.lock = l$
$MkActWaitDoV(l) \triangleq \text{CHOOSE } a \in ActWaitDoV : a.lock = l$

---

Type and initial value of an array of held locks.

$Hold \triangleq [Lock \rightarrow \text{BOOLEAN}]$
$InitHold \triangleq [lock \in Lock \mapsto \text{FALSE}]$ no locks held

---

NODE STATE

Node state for a zero node.

*buf* buffers cars from one train to the next. Everything from the head of *buf* up to but not including a *token* car is part of the current train. Everything from a *token* car on back is part of the next train.

$ZeroNodeState \triangleq$ type definition
[
  $id : ZeroNode,$ the *id* of this node
  $out : Msg,$ sending on ring
  $buf : Seq(Msg)$ buffer cars from one train to the next
]

$InitZeroNodeState(n) \triangleq$ initial value
$[$
  $id \mapsto n,$
  $out \mapsto MkMsgToken,$ initially sending a token $msg$
  $buf \mapsto \langle \rangle$ empty buffer
$]$

$MsgsInZeroNodeState(ns) \triangleq$ set of all messages
  $\{ns.out\} \cup \{ns.buf[x] : x \in 1 \,..\, Len(ns.buf)\}$

$CountTokensInZeroNodeState(ns) \triangleq$ number of tokens stored
  LET
    $Cnt(m) \triangleq$ IF $m \in MsgToken$ THEN $1$ ELSE $0$
  IN
    $Cnt(ns.out) + Sum([x \in 1 \,..\, Len(ns.buf) \mapsto Cnt(ns.buf[x])])$

Node state for a user node.

$UserNodeState \triangleq$ type definition
$[$
  $id : UserNode,$ the $id$ of this node
  $out : Msg,$ sending on ring
  $intrain :$ BOOLEAN , token arrived but not yet idle
  $act : Act,$ current activity of this node
  $hold : Hold,$ locks held by this node
  $didv : Hold$ locks released but not yet sent a $didv$
$]$

$InitUserNodeState(n) \triangleq$ initial value
$[$
  $id \mapsto n,$
  $out \mapsto MkMsgIdle,$ sending idle $msg$
  $intrain \mapsto$ FALSE, not in a train
  $act \mapsto MkActIdle,$ currently idle
  $hold \mapsto InitHold,$ all locks free
  $didv \mapsto InitHold$ all locks free
$]$

$MsgsInUserNodeState(ns) \triangleq$ set of all messages
  $\{ns.out\}$

$CountTokensInUserNodeState(ns) \triangleq$ number of tokens stored

```
LET
  $Cnt(m) \triangleq$ IF $m \in MsgToken$ THEN $1$ ELSE $0$
IN
  $Cnt(ns.out)$
```

Node state for a junction node.

```
$JuncNodeState \triangleq$    type definition
[
  $id : JuncNode,$    the $id$ of this node
  $out : Msg,$    sending on ring
  $intrain :$ BOOLEAN ,    token arrived but not yet idle

  $subout : Msg,$    subring: sending on ring
  $subintrain :$ BOOLEAN ,    subring: token arrived but not yet idle
  $subhold : Hold,$    subring: locks held by any node on subring

  $localtour :$ BOOLEAN    sending real train through subring
]

$InitJuncNodeState(n) \triangleq$    initial value
[
  $id \mapsto n,$
  $out \mapsto MkMsgIdle,$    sending idle $msg$
  $intrain \mapsto$ FALSE,    not in a train

  $subout \mapsto MkMsgIdle,$    sending idle $msg$
  $subintrain \mapsto$ FALSE,    not in a train
  $subhold \mapsto InitHold,$    all locks free

  $localtour \mapsto$ FALSE    not sending real train through subring
]

$MsgsInJuncNodeState(ns) \triangleq$    all messages in a junction node state
  $\{ns.out,\ ns.subout\}$

$CountTokensInJuncNodeState(ns) \triangleq$    number of tokens stored
  LET
    $Cnt(m) \triangleq$ IF $m \in MsgToken$ THEN $1$ ELSE $0$
  IN
    $Cnt(ns.out) + Cnt(ns.subout)$
```

General node state.

```
$NodeState \triangleq \{\}$    type definition
  $\cup\ ZeroNodeState$
  $\cup\ UserNodeState$
  $\cup\ JuncNodeState$
```

```
$InitNodeState(n) \triangleq$    initial value
  IF $n \in ZeroNode$ THEN $InitZeroNodeState(n)$ ELSE
  IF $n \in UserNode$ THEN $InitUserNodeState(n)$ ELSE
  IF $n \in JuncNode$ THEN $InitJuncNodeState(n)$ ELSE
  $Assert($FALSE, "impossible"$)$
```

```
$MsgsInNodeState(ns) \triangleq$    set of all messages
  IF $ns.id \in ZeroNode$ THEN $MsgsInZeroNodeState(ns)$ ELSE
  IF $ns.id \in UserNode$ THEN $MsgsInUserNodeState(ns)$ ELSE
  IF $ns.id \in JuncNode$ THEN $MsgsInJuncNodeState(ns)$ ELSE
  $Assert($FALSE, "impossible"$)$
```

```
$CountTokensInNodeState(ns) \triangleq$    number of tokens stored
  IF $ns.id \in ZeroNode$ THEN $CountTokensInZeroNodeState(ns)$ ELSE
  IF $ns.id \in UserNode$ THEN $CountTokensInUserNodeState(ns)$ ELSE
  IF $ns.id \in JuncNode$ THEN $CountTokensInJuncNodeState(ns)$ ELSE
  $Assert($FALSE, "impossible"$)$
```

---

Given the current *intrain* value $i$, compute the new *intrain* value after the arrival of message $m$.

```
$UpdateIntrain(i, m) \triangleq$
  IF $m \in MsgToken$ THEN TRUE ELSE    start of train
  IF $m \in MsgIdle$ THEN FALSE ELSE    end of train
  $i$
```

---

The set of messages the zero node should discard, rather than recycle.

```
$ZeroNodeDiscardMsgs \triangleq \{\}$
  $\cup\ MsgIdle$
  $\cup\ MsgNull$
```

```
$\cup$ $MsgGotP$
$\cup$ $MsgDidV$
```

On a zero node, append an arriving message $m$ to the buffer $buf$ if $m$ should be recycled.

```
$ZeroNodeAppendBuf(buf, m) \triangleq$
  IF $m \in ZeroNodeDiscardMsgs$ THEN $buf$ ELSE $Append(buf, m)$
```

Compute the new node state resulting from receiving the indicated message on the zero node.

A train consists of consecutive slots starting with a token and ending with an idle.

The zero node is in charge of starting the train. It recycles cars from the previous train in $buf$ . When $buf$ is nonempty, the head of $buf$ is a $token$ and the arriving message is an $idle$ , then it is possible to start a new train.

```
$ZeroNodeRecvMsg(ns, msg) \triangleq$
  IF $ns.id \notin ZeroNode$ THEN $Assert$(FALSE, “not a zero node”)
   ELSE
  IF
      $\wedge$ $ns.buf \neq \langle\rangle$
      $\wedge$ $Head(ns.buf) \in MsgToken$
      $\wedge$ $msg \in MsgIdle$
   THEN
    Start a new train.
    $[ns$ EXCEPT
    $!.out = Head(ns.buf)$,
    $!.buf = ZeroNodeAppendBuf(Tail(ns.buf), msg)$
    $]$
   ELSE
  IF
      $\wedge$ $ns.buf \neq \langle\rangle$
      $\wedge$ $Head(ns.buf) \notin MsgToken$
   THEN
    Continue sending the current train.
    $[ns$ EXCEPT
    $!.out = Head(ns.buf)$,
    $!.buf = ZeroNodeAppendBuf(Tail(ns.buf), msg)$
    $]$
   ELSE
    Send an $idle$ between trains.
    $[ns$ EXCEPT
    $!.out = MkMsgIdle$,
    $!.buf = ZeroNodeAppendBuf(ns.buf, msg)$
    $]$
```

Tell if a user node wants a train.

A user node wants a train if it is not $idle$ or if it has $didv$'s to send.

```
$UserNodeWantsTrain(ns) \triangleq$
  IF $ns.id \notin UserNode$ THEN $Assert(\text{FALSE}, \text{“not a user node”})$
   ELSE
      $\lor ns.act \notin ActIdle$
      $\lor \exists lock \in Lock : ns.didv[lock]$
```

Compute the new node state resulting from receiving the indicated message on a user node.

```
$UserNodeRecvMsg(ns, msg) \triangleq$
  IF $ns.id \notin UserNode$ THEN $Assert(\text{FALSE}, \text{“not a user node”})$
   ELSE
  LET
      Update intrain based on the message being sent. We apply this operator in every case below.
      $ITR(ns0) \triangleq [ns0 \text{ EXCEPT } !.intrain = UpdateIntrain(@, ns0.out)]$

  IN
  IF $\land ns.act \in ActSendReqP$
   $\land msg \in MsgIdle$
   $\land ns.intrain$
   THEN
      We want to send a reqp and here is the idle at the end of the train that we can rewrite to send it.
      $ITR([ns \text{ EXCEPT}$
      $!.out = MkMsgReqP(ns.act.lock, ns.id),$
      $!.act = MkActWaitReqP(ns.act.lock)$
      $])$
   ELSE
  IF $\land ns.act \in ActSendDoV$
   $\land msg \in MsgIdle$
   $\land ns.intrain$
   THEN
      We want to send a dov and here is the idle at the end of the train that we can rewrite to send it.
      $ITR([ns \text{ EXCEPT}$
      $!.out = MkMsgDoV(ns.act.lock, ns.id),$
      $!.act = MkActWaitDoV(ns.act.lock)$
      $])$
   ELSE
  IF $\land \exists lock \in Lock : ns.didv[lock]$
   $\land msg \in MsgIdle$
   $\land ns.intrain$
   THEN
```

We want to send a $didv$ and here is the $idle$ at the end of the train that we can rewrite to send it.

```
  LET $lock \triangleq$ CHOOSE $lock \in Lock : ns.didv[lock]$ IN
  $ITR([ns$ EXCEPT
  $!.out = MkMsgDidV(lock, ns.id)$,
  $!.didv[lock] = $ FALSE
  $])$
ELSE
IF $\land msg \in MsgReqP$
$\land msg.source = ns.id$

$\land Assert(ns.act = MkActWaitReqP(msg.lock)$, “unexpected return reqp”)
$\land Assert(ns.intrain$, “shadow return reqp”)
$\land Assert(\neg ns.hold[msg.lock]$, “return reqp but hold[lock]”)
$\land Assert(\neg ns.didv[msg.lock]$, “return reqp but didv[lock]”)
THEN
```

Our $reqp$ returns. This can only happen when we are waiting for it. The message must be on a real train, not a shadow train.

We own the lock. Rewrite our $reqp$ with a $gotp$ so that our junction node finds out we got the lock. The junction node uses this information to fail subsequent $reqp$'s that may appear on bypass trains.

The return $reqp$ must be on a real train for several reasons. (1) We have a real time slot in which to claim the lock. (2) We can remove the circling $reqp$ by rewriting it to a car type that will not recycle. (3) The resulting $gotp$ will get back to our junction node and update its information before any subsequent bypass train can go past.

```
  IF $BugOmitClaimLock$ THEN
    $ITR([ns$ EXCEPT
    $!.out = MkMsgNull$,
    $!.act = MkActIdle$,
    $!.hold[msg.lock] = @$
    $])$
  ELSE
    $ITR([ns$ EXCEPT
    $!.out = MkMsgGotP(msg.lock, ns.id)$,
    $!.act = MkActIdle$,
    $!.hold[msg.lock] = $ TRUE
    $])$
ELSE
IF $\land msg \in MsgFailP$
$\land msg.source = ns.id$

$\land Assert(ns.act = MkActWaitReqP(msg.lock)$, “unexpected return failp”)
$\land Assert(ns.intrain$, “shadow return failp”)
$\land Assert(\neg ns.hold[msg.lock]$, “return failp but hold[lock]”)
$\land Assert(\neg ns.didv[msg.lock]$, “return failp but didv[lock]”)
THEN
```

Our $reqp$ came back failed. This can only happen when we are are waiting for it. The message must be on a real train, not a shadow train.

We failed to get the lock. Nullify the *failp* .

$ITR([ns$ EXCEPT
$!.out = MkMsgNull,$
$!.act = MkActIdle$
$])$

ELSE

IF $\land\ msg \in MsgReqP$
$\land\ msg.source \neq ns.id$
$\land\ ns.hold[msg.lock]$

$\land\ BugOmitCheckReqP \Rightarrow$ FALSE

$\land\ Assert(ns.intrain,$ “hold[lock] but shadow reqp”)

THEN

Some one else’s *reqp* for a lock we hold. Fail it.

If it is a shadow train, it would be ineffective for us to fail it, because the bypass train already passed our junction node. To handle this case, our junction node is supposed to know about the locks we hold and fail the *reqp* for us on bypass trains, so we should never see it on one. Hence the assert.

$ITR([ns$ EXCEPT $!.out = MkMsgFailP(msg.lock,\ msg.source)])$

ELSE

IF $\land\ msg \in MsgReqP$
$\land\ msg.source \neq ns.id$
$\land\ ns.didv[msg.lock]$

$\land\ BugOmitDidvCheckReqP \Rightarrow$ FALSE

$\land\ Assert(ns.intrain,$ “didv[lock] but shadow reqp”)

THEN

Some one else’s *reqp* for a lock we recently released but have not yet managed to send a *didv* about. Fail it.

If it is a shadow train, it would be ineffective for us to fail it, because the bypass train already passed our junction node. To handle this case, our junction node is supposed to know about the locks we hold and fail the *reqp* for us on bypass trains, so we should never see it on one. Hence the assert.

This is an exceedingly tricky case. It could be that the *reqp* was created by a user node following us on our subring. If we let it go through without failing it, then that user node will claim the lock and rewrite the *reqp* to a *gotp* in order to inform our common junction node that the lock is held on the subring. But meanwhile we are are waiting to send a *didv* to inform the junction node that we have released the lock. Since we add the *didv* to the end of the train, it would arrive at the junction node after the *gotp* , which would mean that the junction node would see a *gotp* for a lock that it thought was already held by a subring user node.

$ITR([ns$ EXCEPT $!.out = MkMsgFailP(msg.lock,\ msg.source)])$

ELSE

IF $\land\ msg \in MsgDoV$
$\land\ msg.source = ns.id$

$\land\ BugContinuousDoV \Rightarrow$ FALSE

$\land\ Assert(ns.act = MkActWaitDoV(msg.lock),$ “unexpected return dov”)
$\land\ Assert(ns.intrain,$ “shadow return dov”)
$\land\ Assert(\neg ns.hold[msg.lock],$ “return dov but hold[lock]”)

```
$\land\ Assert(\neg ns.didv[msg.lock],$ “return dov but didv[lock]”$)$
THEN
```

Our *dov* returns. This can only happen when we are waiting for it. The message must be on a real train, not a shadow train.

Nullify the message.

```
  $ITR([ns$ EXCEPT
  $!.out = MkMsgNull,$
  $!.act = MkActIdle$
  $])$
ELSE
IF $\land\ msg \in MsgDoV$
$\land\ msg.source \neq ns.id$
$\land\ ns.hold[msg.lock]$

$\land\ BugOmitCheckDoV \Rightarrow$ FALSE
$\land\ BugOmitCheckShadowDoV \Rightarrow ns.intrain$

$\land\ Assert(\neg ns.didv[msg.lock],$ “double didv”$)$
THEN
```

Some one else’s *dov* for a lock we hold. Clear our lock and remember to send a *didv* for it, so that our junction node finds out that we cleared our lock.

Note: this can be on a real train or on a shadow train.

We cannot rewrite the *dov* as *null* for several reasons. (1) The *dov* might be on a shadow train, for which rewriting is not permitted by design. (2) The original sender is waiting for the *dov* to return.

```
  $ITR([ns$ EXCEPT
  $!.out = msg,$
  $!.hold[msg.lock] =$ FALSE,
  $!.didv[msg.lock] =$ TRUE
  $])$
ELSE
```

Pass the message unchanged.

```
  $ITR([ns$ EXCEPT $!.out = msg])$
```

---

Compute the new node state resulting from receiving the indicated message on a junction node. The junction node also needs to know what message it is receiving from its subring (*submsg*) and whether any of its subring user nodes want to get the real train (*wantreal*).

```
$JuncNodeRecvMsg(ns, msg, submsg, wantreal) \triangleq$
  IF $ns.id \notin JuncNode$ THEN $Assert($FALSE, “not a junction node”$)$
  ELSE
  LET
```

If we are in local tour mode and a real train is coming out of our subring, then we have to update our *subhold* based on the message coming out. Compute our new *subhold* for this case.

```
newsubhold ≜
  LET
    h ≜ ns.subhold      our current subhold
    m ≜ submsg      the message coming out of our subring
    mine ≜ m.source[1] = ns.id[1]    was created on our subring
  IN
    IF m ∈ MsgGotP ∧ mine THEN [h EXCEPT ![m.lock] = TRUE] ELSE
    IF m ∈ MsgDidV ∧ mine THEN [h EXCEPT ![m.lock] = FALSE] ELSE
    h

If we are processing a train in bypass mode, then we have to fail any reqp that conflicts with a lock held by
our subring. Compute the output message for this case.
bypassmsg ≜
  LET
    h ≜ ns.subhold      our current subhold
    m ≜ msg      the message coming along the main ring
    held ≜ h[m.lock]     our subring holds the lock
  IN
    IF m ∈ MsgReqP ∧ held THEN MkMsgFailP(m.lock, m.source) ELSE
    m

IN
IF
   ∧ msg ∈ MsgToken
   ∧ wantreal

   ∧ Assert(¬ns.subintrain, "multiple real trains")
   ∧ Assert(¬UpdateIntrain(ns.subintrain, submsg), "multiple real trains")
 THEN
  Train arrives and our subring wants a real train.
  [ns EXCEPT
  !.out = MkMsgIdle,
  !.subout = msg,               send the real train down subring
  !.localtour = TRUE,              sending the train on a local tour
  !.intrain = UpdateIntrain(@, msg),
  !.subintrain = UpdateIntrain(@, submsg)
  ]
 ELSE
IF
   ∧ msg ∈ MsgToken
   ∧ ¬wantreal

   ∧ Assert(¬ns.localtour, "multiple real trains")
   ∧ Assert(¬ns.subintrain, "multiple real trains")
   ∧ Assert(¬UpdateIntrain(ns.subintrain, submsg), "multiple real trains")
 THEN
```

```
  Train arrives and our subring would be happy with a shadow train.
  [$ns$ EXCEPT
  $!.out = msg$,      bypass the real train
  $!.subout = MkMsgIdle$,      shadow train does not have a token
  $!.localtour = \text{FALSE}$,      sending the train on a bypass
  $!.intrain = UpdateIntrain(@, msg)$,
  $!.subintrain = UpdateIntrain(@, submsg)$
  ]
 ELSE
IF
    $\land UpdateIntrain(ns.subintrain, submsg)$

    $\land Assert(ns.localtour, \text{“wrong configuration”})$
 THEN
  Train is coming out of subring. This can only happen if we are sending the train on a local tour, hence the
  assert.

  This is the only case in which we update our $subhold$.
  [$ns$ EXCEPT
  $!.out = submsg$,      train comes out
  $!.subout = msg$,      train (if any) going in
  $!.intrain = UpdateIntrain(@, msg)$,
  $!.subintrain = UpdateIntrain(@, submsg)$,
  $!.subhold = newsubhold$
  ]
 ELSE
IF
    $\land ns.subintrain$
    $\land \neg UpdateIntrain(ns.subintrain, submsg)$

    $\land Assert(ns.localtour, \text{“wrong configuration”})$
    $\land Assert(msg \in MsgIdle, \text{“impossible finish”})$
    $\land Assert(submsg \in MsgIdle, \text{“impossible finish”})$
    $\land Assert(\neg ns.intrain, \text{“impossible trains”})$
 THEN
  Train just finished coming out of subring.
  [$ns$ EXCEPT
  $!.out = MkMsgIdle$,
  $!.subout = MkMsgIdle$,
  $!.localtour = \text{FALSE}$,      end of the local tour
  $!.intrain = UpdateIntrain(@, msg)$,
  $!.subintrain = UpdateIntrain(@, submsg)$
  ]
 ELSE
IF
    $\land ns.localtour$
    $\land$ $Assert(\neg ns.subintrain,$ "impossible trains")
    $\land$ $Assert(\neg UpdateIntrain(ns.subintrain,\ submsg),$ "impossible trains")
 THEN
  Train is or has gone into subring, but is not coming out yet.
  $[ns$ EXCEPT
  $!.out = MkMsgIdle,$                nothing coming out yet
  $!.subout = msg,$                train (if any) going in
  $!.intrain = UpdateIntrain(@,\ msg),$
  $!.subintrain = UpdateIntrain(@,\ submsg)$
  $]$
 ELSE
IF
    $\land \neg ns.localtour$

    $\land$ $Assert(\neg ns.subintrain,$ "impossible trains")
    $\land$ $Assert(\neg UpdateIntrain(ns.subintrain,\ submsg),$ "impossible trains")
 THEN
  If there is a train, we are processing it in bypass mode.
  $[ns$ EXCEPT
  $!.out = bypassmsg,$
  $!.subout = bypassmsg,$
  $!.intrain = UpdateIntrain(@,\ msg),$
  $!.subintrain = UpdateIntrain(@,\ submsg)$
  $]$
 ELSE
  $Assert($FALSE, "impossible")
```

STATE

```
$State \triangleq [$
  $nodes : [Node \rightarrow NodeState]$
$]$

$InitState \triangleq [$
  $nodes \mapsto [n \in Node \mapsto InitNodeState(n)]$
$]$
```

NEXT STATE RELATIONS

Advance by one step.

Each node receives the message sent by the node to its left, and updates its state, computing the message it sends in the next step.

```
$NextStep \triangleq$
  LET
    $NN \triangleq state.nodes$

    Given a node $id$ $n$, find the node $id$ of the previous node on the main ring.
    $PrevMainRingNode(n) \triangleq$
      CHOOSE $fn \in Node$ :
        $\land (fn[1] + 1)\%(Len(NumNode) + 1) = n[1]$   prev on main ring
        $\land Len(fn) = 2 \Rightarrow fn[2] = 0$   junc if a subring

    Given a node $id$ $n$, compute the node $id$ of the previous node on the same subring.
    $PrevSubringNode(n) \triangleq$
      CHOOSE $fn \in Node$ :
        $\land Len(fn) = 2$   on a subring
        $\land fn[1] = n[1]$   same as $n$
        $\land (fn[2] + 1)\%(NumNode[n[1]] + 1) = n[2]$   prev position

    How to update a zero node.
    $zeroupd(n) \triangleq$
      LET
        $ns \triangleq NN[n]$
        $msg \triangleq NN[PrevMainRingNode(n)].out$
      IN
        $ZeroNodeRecvMsg(ns, msg)$

    How to update a user node.
    $userupd(n) \triangleq$
      LET
        $ns \triangleq NN[n]$
        $msg \triangleq$ IF $Len(n) = 1$ THEN $NN[PrevMainRingNode(n)].out$
         ELSE
           LET $fn \triangleq PrevSubringNode(n)$IN
           IF $fn[2] = 0$ THEN $NN[fn].subout$ ELSE $NN[fn].out$
      IN
        $UserNodeRecvMsg(ns, msg)$

    How to update a junction node.
    $juncupd(n) \triangleq$
      LET
        $ns \triangleq NN[n]$
        $msg \triangleq NN[PrevMainRingNode(n)].out$
        $submsg \triangleq NN[PrevSubringNode(n)].out$
        $wantreal \triangleq \exists\, un \in UserNode :$
          $\land\ un[1] = n[1]$   same subring
          $\land\ UserNodeWantsTrain(NN[un])$   wants a train
      IN
        $JuncNodeRecvMsg(ns,\ msg,\ submsg,\ wantreal)$
  IN
    $state' =$
    $[state$ EXCEPT
    $!.nodes =$
    $[n \in Node \mapsto$
    IF $n \in ZeroNode$ THEN $zeroupd(n)$ ELSE
    IF $n \in UserNode$ THEN $userupd(n)$ ELSE
    IF $n \in JuncNode$ THEN $juncupd(n)$ ELSE
    $Assert($FALSE, "uncovered"$)$
    $]$
    $]$
```

Some user node decides to try to take a lock it does not hold.

```
$NextTakeUnheldLock \triangleq$
  $\exists\, node \in UserNode :$
  $\exists\, lock \in Lock :$
  LET
    $ns \triangleq state.nodes[node]$
  IN
     $\land \neg UserNodeWantsTrain(ns)$   must not want train
     $\land \neg ns.hold[lock]$   lock not currently held
     $\land\ state' =$
    $[state$ EXCEPT $!.nodes[node] =$
    $[@$ EXCEPT
    $!.act = MkActSendReqP(lock)$
    $]$
    $]$
```

Some user node decides to release a lock it holds.

```
$NextReleaseHeldLock \triangleq$
  $\exists\, node \in UserNode :$
  $\exists\, lock \in Lock :$
```

```
LET
  $ns \triangleq state.nodes[node]$
IN
   $\land \neg UserNodeWantsTrain(ns)$          must not want train
   $\land ns.hold[lock]$          lock currently held

   $\land state' =$
  $[state$ EXCEPT $!.nodes[node] =$
  $[@$ EXCEPT
  $!.hold[lock] =$ FALSE,
  $!.didv[lock] =$ IF $BugOmitDidV$ THEN $@$ ELSE
  TRUE
  $]$
  $]$
```

Some user node decides to release a lock it does not hold.

```
$NextReleaseUnheldLock \triangleq$
  $\exists node \in UserNode :$
  $\exists lock \in Lock :$
  LET
    $ns \triangleq state.nodes[node]$
  IN
     $\land \neg UserNodeWantsTrain(ns)$          must not want train
     $\land \neg ns.hold[lock]$          lock not currently held
     $\land state' =$
    $[state$ EXCEPT $!.nodes[node] =$
    $[@$ EXCEPT
    $!.act = MkActSendDoV(lock)$
    $]$
    $]$
```

---

INVARIANTS

The state must always be of the proper type.

```
$InvType \triangleq$
   $\land state \in State$
```

There is always exactly one token.

```
$InvUniqueToken \triangleq$
  LET
    Total count of tokens stored over all nodes.
    $total \triangleq Sum([n \in Node \mapsto CountTokensInNodeState(state.nodes[n])])$
  IN
    $total = 1$
```

No two user nodes can hold the same lock.

```
$InvLockMutex \triangleq$
  $\forall n1, n2 \in UserNode :$
  $\forall lock \in Lock :$
  $($
     $\land state.nodes[n1].hold[lock]$
     $\land state.nodes[n2].hold[lock]$
  $) \Rightarrow n1 = n2$
```

When a junction node is not in local tour mode, it is correct about what locks are (effectively) held by its subring user nodes.

```
$InvBypassSubhold \triangleq$
  LET
    Compute what the $subhold$ of junction node $jn$ for lock $k$ should be, assuming that the junction node is not in local tour mode.
    $calcsubhold(jn, k) \triangleq$
      $\exists un \in UserNode :$
        $\land un[1] = jn[1]$    same subring and
        $\land \lor state.nodes[un].hold[k]$    either lock held
        $\lor state.nodes[un].didv[k]$    or waiting to send $didv$

  IN
  $\forall jn \in JuncNode :$
  $\neg state.nodes[jn].localtour \Rightarrow$
   $\forall lock \in Lock :$
   $state.nodes[jn].subhold[lock] = calcsubhold(jn, lock)$
```

TEMPORAL ASSUMPTIONS

Liveness assumption.

We have to make explicit the temporal assumption that always eventually a time step happens on the ring. (Otherwise, the model permits infinite suttering.) Since the step is always enabled, weak fairness is sufficient.

```
$Liveness \triangleq$
  $\land \mathbf{WF}_{state}(NextStep)$
```

TEMPORAL PROPERTIES

For all user nodes, the node always eventually gets back to the idle state.

```
$RequestCompletion \triangleq$
  $\forall n \in UserNode :$
  LET
    This user node is idle.
    $idle \triangleq state.nodes[n].act \in ActIdle$
  IN
    $\Box\Diamond idle$
```

For all locks, always, if the lock is always eventually desired, then eventually the lock is held.

```
$LockAcquisition \triangleq$
  $\forall lock \in Lock :$
  LET
    Some node desires this lock.
    $desired \triangleq \exists n \in UserNode :$
    $state.nodes[n].act = MkActSendReqP(lock)$

    Some node holds this lock.
    $held \triangleq \exists n \in UserNode :$
       $state.nodes[n].hold[lock]$
  IN
    $\Box(\Box\Diamond desired \Rightarrow \Diamond held)$
```

For all locks, some node wanting to $dov$ the lock leads to no node holding it.

```
$LockForceReleasing \triangleq$
  $\forall lock \in Lock :$
  LET
    Some node wants to $dov$ this lock.
```

$wantdov \triangleq \exists\, n \in UserNode :$
$state.nodes[n].act = MkActSendDoV(lock)$

No node holds this lock.

$free \triangleq \forall\, n \in UserNode :$
$\quad \neg state.nodes[n].hold[lock]$

IN

$wantdov \leadsto free$

---

SPECIFICATION

Initial state.

$Init \triangleq$
$\quad \land\ state = InitState$

Next state relation.

$Next \triangleq$
$\quad \lor\ NextStep$
$\quad \lor\ NextTakeUnheldLock$
$\quad \lor\ NextReleaseHeldLock$
$\quad \lor\ NextReleaseUnheldLock$

Specification.

$Spec \triangleq$
$\quad \land\ Init$
$\quad \land\ \Box[Next]_{state}$
$\quad \land\ Liveness$

THEOREM $Spec \Rightarrow$
$\quad \land\ \Box InvType$
$\quad \land\ \Box InvUniqueToken$
$\quad \land\ \Box InvLockMutex$
$\quad \land\ \Box InvBypassSubhold$
$\quad \land\ RequestCompletion$
$\quad \land\ LockAcquisition$
$\quad \land\ LockForceReleasing$