Models of Software Systems **Name: Daniel Pagan**
Independent Project: Hazard Pointers **Due: 1 December 2021**

1 Hazard Pointer Modeling

1.1 Problem

With the slowing of Moore's Law, computing has set off in search of ways to satiate our growing need for computing power. Moving away from monolithic systems that operate on a single core without comunicating with the outside world has led us to find a better way to scale: distributed computing. Distributed computing relies on many computation objects to solve our problems in parallel (whether it be machines, cores, or hyperthreading). In order to efficiently comunicate and use these kinds of systems, we have developed data structures where many readers and writers can concurrently share the same structure.

When going from normal data structures to concurrent ones, we run into many synchronization issues. Reading from a memory locaiton is unstable because a concurrent writer could be shifting the data out from under us. If we read a value, it could instantly be invalidated before we can perform some meaningful computation on it. We also care about the order of things. If we read something, we want to perform our computation and write an update back while preventing others from computing on old data. All of these problems must be addressed in order for users of a concurrent data structure to build their business logic and be able to reason about their code with the level of abstractions that a single threaded environment provides them. This is especially important for abstracting away business logic from computing semantics in a complex system.

One solution to the problem of concurrent reading and writing is through locking. A lock is simply an atomic marker that lets only one thread access a data structure at a time. This serializes reads and writes, solving our initial problem. A lock has the idea of acquire and release operations which protect the underlying data. Locking structures have "critical sections" of the code where the lock is acquired and all other concurrent processes must wait until the lock is released.

Locking leads to many problems for performant systems. A system that uses a complex data structure like an AVL tree might use a global lock to serialize reads and writes. This would work in theory, but when put to the test, the system breaks down with contention on the lock. A common scenario is for many readers to want access to the system while only a couple of writers try to update it. For the many readers, they all must wait for each other while the underlying data may not even be changing. One way to solve this is to have granular locking, which places many locks on individual subsets of the data structure. This serializes access but requires deep tracking of the locking structures and also has a slight memory overhead for keeping track of all of the locks. Even still, granular locking still falls prey to contention with readers. An example of this would be a concurrent list where 20 readers are all trying to read the head of the list. No ammount of granularization would assist in this case.

Locking leads to deadlock, which is where a thread cannot proceed because the lock is taken indefinitely. It also leads to interesting edge cases where a thread that dies with a held lock leaves the lock forever deadlocked for other threads.

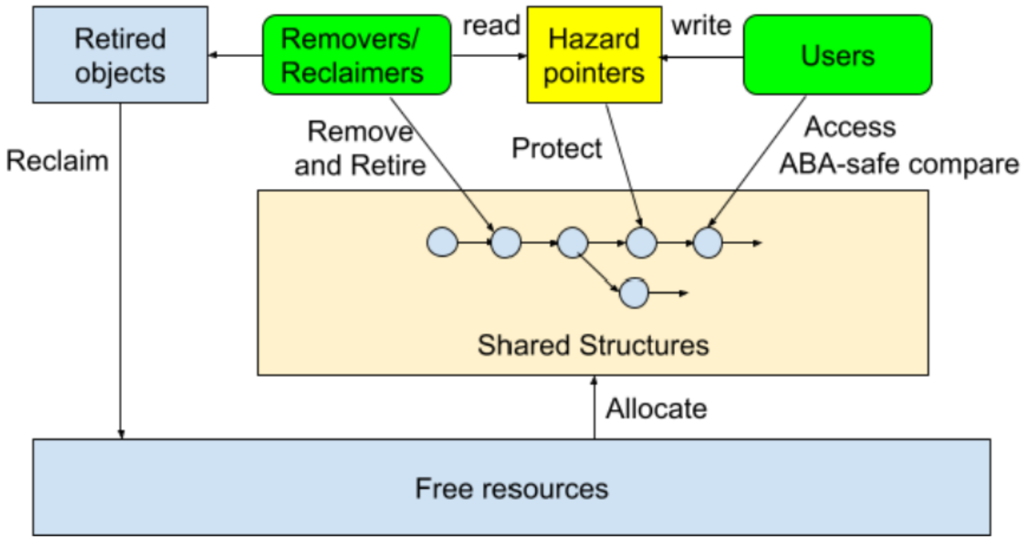
These issues have led to the invention of lock-free concurrent data structures. They attempt to prevent contention by removing locks all-together. This is achieved through atomic instructions that can swap values based on what is already there. This is called a compare and set instruction. Usually these are used to compare and set pointers to the data structures they are protecting.

Lock-free algorithms are appealing because they allow for truly concurrent reads and atomic writes. This means that readers can keep reading without delay if the value underneath has not changed.

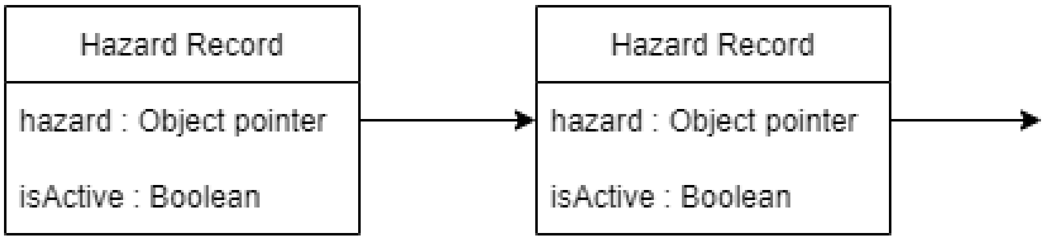
One problem that comes up in the lock-free world is the ABA problem. It based on a reader reading a value A and performing some meaningful computation on the value. Another thread comes along and changes the value to B and then back to A underneath of the reader thread. This tricks the original thread into thinking it still has the most current value even though there have been hidden writes underneath of it. This shows that while lock-free algorithms yield better performance, they are dangerous because they give rise to new forms problems.

All of this has led up to the main topic of this paper: hazard pointers. A hazard pointer is a lock-free, wait-free primitive to wrap around a data structure that is optimized for many readers and rare writes. It was introduced in 2004 by Andrei Alexandrescu and Maged Michael. It improves upon previous compare and set algorithms by providing a solidly defined solution to the ABA problem. Facebook has recently used hazard pointers in its open source repository of C++ data structures and algorithms, folly.

Hazard pointers almost keep track of epochs of models of the underlying data structure. Every write produces a new copy of the data structure. Hazard pointers model readers as users which keep track of which pointers are currently in use (a single hazard pointer is an active pointer to a copy of the model at a point in time). Writers are modeled as removers/reclaimers, which are able to sort through all of the hazard pointers that are valid and reclaim the memory of those that are not active and are no longer referenced. This is very similar to epoch based garbage collection except we get ABA-safe readers and better interleaving of operations than a full garbage collected system. The following image describes the interaction:



A full hazard pointer structure is made up of global state and per-thread state. The global state includes a linked list of all of the hazard records. Each hazard record is used to track a hazard pointer and its status as being active or inactive. A hazard pointer being active is just a claim to not remove that particular instance of the data structure. The following image shows a diagram of the structure:



The hazard record list is actually not needed to be a linked list. It is simply a set of active hazard pointers and could be optimized in a different implementation. We will see later in my modeling how this quirk becomes useful in abstracting this structure.

Each thread labeled as a writer has both the writing and reclamation responsibilities. The writing is done by allocating a new copy of the structure with the desired update. The reclamation is done by looping through any structures marked as no longer needed by readers (hazard records that are active) and performing a set difference with the currently allocated structures. In order to achieve this, the writers need to track revocations for later reclamation. This is done through a thread local structure called the revocation list. It is simply a set of hazard pointers that have been marked for reclamation.

Hazard pointers attack the ABA problem by limiting readers to their own worlds. Readers get a pointer to a structure that they can freely read from without worrying about other writes coming in on top. One thing that readers cannot do is write to their returned pointer's structure. Readers from the same epoch share the same underlying structure. This also means that multiple hazard records can point to the same underlying structure (writers must wait for all readers to be done with a structure to reclaim the memory).

1.2 Approach

I chose TLA+ to model my system. I think that is lends itself to this particular challenge better than any other modeling system. TLA+ itself allows for global state machine logic which is actually not very suited to our problem. We want to model concurrent processes that do not respect each other to allow them to have execution time (unfair). Luckily, PlusCal adds these constructs on top of TLA+.

I used PlusCal's concurrency model to create separate concurrent processes for the readers and writers. PlusCal by default creates unfair processes that do not wait for each other. These features are added in as keywords, so my logic is that if I can get a valid hazard pointer model written in PlusCal that never uses *await* or *fair*, then I have proved that the algorithm is wait free. Another goal is to prove that the algorithm never halts which is built into the TLC checker with deadlock checking. This lets me exhaustively check that hazard pointers will not fail in their normal operation.

This model also lets me use PlusCal's *assert* call to check that there is never a state where a read can be modified underneath of itself. This verifies the logic behind the ABA problem's solution in hazard pointers.

I began by modeling a simple global lock system to be able to compare the number of waits needed in the algorithm. This model simply used a shared mutex to lock the number to write. I verified using *assert* debugging that critical sections were not executed concurrently. Unfortunately, there is no way to incorporate performance modeling into TLA+.

The folly repository uses its hazard pointer implementation to implement a concurrent hash-map. I actually attempted to fully model their implementation of a hazard pointer concurrent hash-map, but the model ended up having too many states to verify. Because of TLA+'s graph-based exhaustive search through the model's state space, I was limited by my machine's resources and I actually was unable to finish evaluation before my laptop ran out of memory. I ended up modeling only a simple integer reading and writing model.

Modeling the simplified integer read/write model ended up being tricky because I had to model pointers within TLA+. There is no concept of a memory address pointer in this language, so I had to come up with a way to model what hazard pointers were achieving in a creative way. I used a timeline based approach with a mapping. By keeping track of each new write as a separate entry in a sequence, I was able to mimic temporal structures within a temporal language. This ended up being equivalent to pointers because every structure is only written once and is then never updated.

One part of the system that I was unable to model was the failure of reader threads. I did not find a simple way to make PlusCal processes crash randomly. Also, it is unclear how to formalize memory reclamation when there is no equivalent to the destruction of objects in logic (they are just no longer in existance once no longer referenced). If a reader thread fails during its active time, it will basically pin a copy of the internal data structure because the active flag will never be unset.

1.3 Results

I was able to successfully model the system based on the abstractions listed above. The model showed that the simple model fully serialized actions while the hazard pointer model safely interleaved reads with writes. I showed that the system did not need to wait by never using the *await* keyword in my model.

This model implies that it is safe to use hazard pointers in their intended use case. It did not, however, prove that failure could not mess up the memory usage in a system. Thread failures could leave memory zombified to dead readers. In theory, memory may run out due to this quirk of the system. From there, the model does not define what will happen.

I think that the end result is a great primitive for efficient, lock-free data structures. I would caution implementers with the edge cases that my modeling cannot verify. The folly repository also leaves this edge case open to the machine the code is running on. This could be dangerous for high assurance systems that need to have every edge case covered.

1.4 Limitations and Future Work

State space explosion is a big problem for sufficiently complex systems. Memory usage is something that is difficult to model without writing an entire model for the memory (and having a massive state space explosion as well as issues with memory limits). I was limited in my ability to represent complex data structures as well.

The ability to kill readers and show the effects on the algorithm was a limitation of the modeling suite. I think future work could improve upon this by attempting to write some form of penalty function for readers that fail. I think it could scale based on the number of readers that are impacted by a single reader failing during its active record state.

One possible extension to the work would be to model the interleaving of hazard pointers with other lock-free primitives. I think that the reclamation methods in hazard pointers could be utilized in creative ways to synergize with lock-free systems that front load work to readers rather than writers.

1.5 Supplementary Information

Simple Lock:

```
---------------------- MODULE simple_lock ----------------------
EXTENDS Naturals, TLC
CONSTANTS NumReaders, NumWrites

  --algorithm simple_lock
variables cur = 0, lock = 0
```

```
process writer = 1
variable write = 0
begin
Writer_Loop:
    while $write \neq NumWrites$ do
        Write_Acquire:
            await $lock = 0$ ;
            $lock := 1$ ;
        Update:
            $cur := write$ ;
        Write_Release:
            $lock := 0$ ;
    end while ;
end process ;

process $reader \in 2 \,..\, (NumReaders + 1)$
variables saved_read
begin
Read_Acquire:
    await $lock = 0$ ;
    $lock := 1$ ;
Read:
    $saved\_read := cur$ ;
Check:
    assert $saved\_read = cur$ ;
Read_Release:
    $lock := 0$ ;
end process ;

end algorithm ;
```

BEGIN TRANSLATION ($chksum(pcal) =$ "1359a2df" $\land\ chksum(tla) =$ "8136cae")

CONSTANT $defaultInitValue$

VARIABLES $cur, lock, pc, write, saved\_read$

$vars \triangleq \langle cur, lock, pc, write, saved\_read \rangle$

$ProcSet \triangleq \{1\} \cup (2 \,..\, (NumReaders + 1))$

$Init \triangleq$ Global variables

$\land\ cur = 0$

$\land\ lock = 0$

Process writer

$\land\ write = 0$

Process reader

$\land\ saved\_read = [self \in 2 \,..\, (NumReaders + 1) \mapsto defaultInitValue]$

$\land\ pc = [self \in ProcSet \mapsto$ CASE $self = 1 \to$ "Writer_Loop"

$\qquad\qquad\qquad\qquad \Box\ self \in 2 \,..\, (NumReaders + 1) \to$ "Read_Acquire"$]$

$$\begin{array}{l}
\mathit{Writer\_Loop} \triangleq \land\ pc[1] = \text{“Writer\_Loop”} \\
\qquad\qquad \land\ \textsc{if}\ \mathit{write} \neq \mathit{NumWrites} \\
\qquad\qquad\qquad \textsc{then}\ \land\ pc' = [pc\ \textsc{except}\ ![1] = \text{“Write\_Acquire”}] \\
\qquad\qquad\qquad \textsc{else}\ \ \land\ pc' = [pc\ \textsc{except}\ ![1] = \text{“Done”}] \\
\qquad\qquad \land\ \textsc{unchanged}\ \langle \mathit{cur}, \mathit{lock}, \mathit{write}, \mathit{saved\_read} \rangle
\end{array}$$

$$\begin{array}{l}
\mathit{Write\_Acquire} \triangleq \land\ pc[1] = \text{“Write\_Acquire”} \\
\qquad\qquad \land\ \mathit{lock} = 0 \\
\qquad\qquad \land\ \mathit{lock}' = 1 \\
\qquad\qquad \land\ pc' = [pc\ \textsc{except}\ ![1] = \text{“Update”}] \\
\qquad\qquad \land\ \textsc{unchanged}\ \langle \mathit{cur}, \mathit{write}, \mathit{saved\_read} \rangle
\end{array}$$

$$\begin{array}{l}
\mathit{Update} \triangleq \land\ pc[1] = \text{“Update”} \\
\qquad\quad \land\ \mathit{cur}' = \mathit{write} \\
\qquad\quad \land\ pc' = [pc\ \textsc{except}\ ![1] = \text{“Write\_Release”}] \\
\qquad\quad \land\ \textsc{unchanged}\ \langle \mathit{lock}, \mathit{write}, \mathit{saved\_read} \rangle
\end{array}$$

$$\begin{array}{l}
\mathit{Write\_Release} \triangleq \land\ pc[1] = \text{“Write\_Release”} \\
\qquad\qquad \land\ \mathit{lock}' = 0 \\
\qquad\qquad \land\ pc' = [pc\ \textsc{except}\ ![1] = \text{“Writer\_Loop”}] \\
\qquad\qquad \land\ \textsc{unchanged}\ \langle \mathit{cur}, \mathit{write}, \mathit{saved\_read} \rangle
\end{array}$$

$$\mathit{writer} \triangleq \mathit{Writer\_Loop} \lor \mathit{Write\_Acquire} \lor \mathit{Update} \lor \mathit{Write\_Release}$$

$$\begin{array}{l}
\mathit{Read\_Acquire}(\mathit{self}) \triangleq \land\ pc[\mathit{self}] = \text{“Read\_Acquire”} \\
\qquad\qquad\qquad \land\ \mathit{lock} = 0 \\
\qquad\qquad\qquad \land\ \mathit{lock}' = 1 \\
\qquad\qquad\qquad \land\ pc' = [pc\ \textsc{except}\ ![\mathit{self}] = \text{“Read”}] \\
\qquad\qquad\qquad \land\ \textsc{unchanged}\ \langle \mathit{cur}, \mathit{write}, \mathit{saved\_read} \rangle
\end{array}$$

$$\begin{array}{l}
\mathit{Read}(\mathit{self}) \triangleq \land\ pc[\mathit{self}] = \text{“Read”} \\
\qquad\qquad \land\ \mathit{saved\_read}' = [\mathit{saved\_read}\ \textsc{except}\ ![\mathit{self}] = \mathit{cur}] \\
\qquad\qquad \land\ pc' = [pc\ \textsc{except}\ ![\mathit{self}] = \text{“Check”}] \\
\qquad\qquad \land\ \textsc{unchanged}\ \langle \mathit{cur}, \mathit{lock}, \mathit{write} \rangle
\end{array}$$

$$\begin{array}{l}
\mathit{Check}(\mathit{self}) \triangleq \land\ pc[\mathit{self}] = \text{“Check”} \\
\qquad\qquad \land\ \mathit{Assert}(\mathit{saved\_read}[\mathit{self}] = \mathit{cur}, \\
\qquad\qquad\qquad\qquad \text{“Failure of assertion at line 32, column 5.”}) \\
\qquad\qquad \land\ pc' = [pc\ \textsc{except}\ ![\mathit{self}] = \text{“Read\_Release”}] \\
\qquad\qquad \land\ \textsc{unchanged}\ \langle \mathit{cur}, \mathit{lock}, \mathit{write}, \mathit{saved\_read} \rangle
\end{array}$$

$$\begin{array}{l}
\mathit{Read\_Release}(\mathit{self}) \triangleq \land\ pc[\mathit{self}] = \text{“Read\_Release”} \\
\qquad\qquad\qquad \land\ \mathit{lock}' = 0 \\
\qquad\qquad\qquad \land\ pc' = [pc\ \textsc{except}\ ![\mathit{self}] = \text{“Done”}] \\
\qquad\qquad\qquad \land\ \textsc{unchanged}\ \langle \mathit{cur}, \mathit{write}, \mathit{saved\_read} \rangle
\end{array}$$

$$\begin{array}{l}
\mathit{reader}(\mathit{self}) \triangleq \mathit{Read\_Acquire}(\mathit{self}) \lor \mathit{Read}(\mathit{self}) \lor \mathit{Check}(\mathit{self}) \\
\qquad\qquad \lor\ \mathit{Read\_Release}(\mathit{self})
\end{array}$$

Allow infinite stuttering to prevent deadlock on termination.

$Terminating \triangleq \land \forall self \in ProcSet : pc[self] = \text{"Done"}$
$\qquad\qquad\qquad\quad \land \text{UNCHANGED } vars$

$Next \triangleq writer$
$\qquad\quad \lor (\exists self \in 2 \,..\, (NumReaders + 1) : reader(self))$
$\qquad\quad \lor Terminating$

$Spec \triangleq Init \land \Box[Next]_{vars}$

$Termination \triangleq \Diamond(\forall self \in ProcSet : pc[self] = \text{"Done"})$

END TRANSLATION

Simple Lock Configuration:

```
SPECIFICATION Spec
CONSTANT defaultInitValue = defaultInitValue
\* Add statements after this line.

CONSTANTS NumReaders = 3
    NumWrites = 3
```

Hazard Pointer:

MODULE *hazptr*

EXTENDS *Naturals*, *TLC*, *FiniteSets*, *Sequences*
CONSTANTS *NumReaders*, *NumWrites*

$R \triangleq 2$

```
  --algorithm hazptr
variables $cur = 0$, $hzdreclist = \{\}$, $timeline = \langle\rangle$

process $writer = 1$
variables $old = 0$, $rlist = \{\}$, $write = 0$
begin
WriterLoop:
   while $write \neq NumWrites$ do
   Update:
      $old := cur$ ;
      $cur := cur + 1$ ;
      $timeline := Append(timeline, write)$ ;
   Retire:
      $rlist := rlist \cup \{old\}$ ;
   Scan:
      if $Cardinality(rlist) \geq R$ then
```

```
            free all ones without hazard pointers
            $rlist := \{x \in rlist : x \in hzdreclist\}$ ;
            print "cleaned rlist" ;
        end if ;
        $write := write + 1$
end while ;
end process ;

process $reader \in 2 \,..\, NumReaders + 1$
variables $saved\_read = 0$, $saved\_read\_ptr = 0$
begin
Acquire:
    await $1 \in \text{DOMAIN } timeline$ ;
      wait until something can be read. This mimics a real function that would
      check for the data to not be null
    $hzdreclist := hzdreclist \cup \{cur\}$ ;
Read:
    $saved\_read\_ptr := cur$ ;
    $saved\_read := timeline[cur]$ ;
Check:
      mimic a pointer dereference
    assert $saved\_read = timeline[saved\_read\_ptr]$ ;
Release:
    $hzdreclist := hzdreclist \setminus \{cur\}$ ;
end process ;

end algorithm ;
```

BEGIN TRANSLATION ($chksum(pcal) =$ "$e300f735$" $\land\ chksum(tla) =$ "$456bb21d$")

VARIABLES cur, $hzdreclist$, $timeline$, pc, old, $rlist$, $write$, $saved\_read$, $saved\_read\_ptr$

$vars \triangleq \langle cur, hzdreclist, timeline, pc, old, rlist, write, saved\_read, saved\_read\_ptr \rangle$

$ProcSet \triangleq \{1\} \cup (2 \,..\, NumReaders + 1)$

$Init \triangleq$ Global variables

$\land\ cur = 0$

$\land\ hzdreclist = \{\}$

$\land\ timeline = \langle\rangle$

Process writer

$\land\ old = 0$

$\land\ rlist = \{\}$

$\land\ write = 0$

Process reader

$\land\ saved\_read = [self \in 2 \,..\, NumReaders + 1 \mapsto 0]$

$\land\ saved\_read\_ptr = [self \in 2 \,..\, NumReaders + 1 \mapsto 0]$

```
        $\land pc = [self \in ProcSet \mapsto \text{CASE } self = 1 \rightarrow \text{"WriterLoop"}$
                                         $\Box \; self \in 2 \,..\, NumReaders + 1 \rightarrow \text{"Acquire"}]$

$WriterLoop \triangleq \land pc[1] = \text{"WriterLoop"}$
                $\land \text{IF } write \neq NumWrites$
                       $\text{THEN } \land pc' = [pc \text{ EXCEPT } ![1] = \text{"Update"}]$
                       $\text{ELSE } \land pc' = [pc \text{ EXCEPT } ![1] = \text{"Done"}]$
                $\land \text{UNCHANGED } \langle cur, hzdreclist, timeline, old, rlist, write,$
                                     $saved\_read, saved\_read\_ptr \rangle$

$Update \triangleq \land pc[1] = \text{"Update"}$
              $\land old' = cur$
              $\land cur' = cur + 1$
              $\land timeline' = Append(timeline, write)$
              $\land pc' = [pc \text{ EXCEPT } ![1] = \text{"Retire"}]$
              $\land \text{UNCHANGED } \langle hzdreclist, rlist, write, saved\_read, saved\_read\_ptr \rangle$

$Retire \triangleq \land pc[1] = \text{"Retire"}$
              $\land rlist' = (rlist \cup \{old\})$
              $\land pc' = [pc \text{ EXCEPT } ![1] = \text{"Scan"}]$
              $\land \text{UNCHANGED } \langle cur, hzdreclist, timeline, old, write, saved\_read,$
                                 $saved\_read\_ptr \rangle$

$Scan \triangleq \land pc[1] = \text{"Scan"}$
            $\land \text{IF } Cardinality(rlist) \geq R$
                  $\text{THEN } \land rlist' = \{x \in rlist : x \in hzdreclist\}$
                        $\land PrintT(\text{"cleaned rlist"})$
                  $\text{ELSE } \land \text{TRUE}$
                        $\land rlist' = rlist$
            $\land write' = write + 1$
            $\land pc' = [pc \text{ EXCEPT } ![1] = \text{"WriterLoop"}]$
            $\land \text{UNCHANGED } \langle cur, hzdreclist, timeline, old, saved\_read,$
                               $saved\_read\_ptr \rangle$

$writer \triangleq WriterLoop \lor Update \lor Retire \lor Scan$

$Acquire(self) \triangleq \land pc[self] = \text{"Acquire"}$
                   $\land 1 \in \text{DOMAIN } timeline$
                   $\land hzdreclist' = (hzdreclist \cup \{cur\})$
                   $\land pc' = [pc \text{ EXCEPT } ![self] = \text{"Read"}]$
                   $\land \text{UNCHANGED } \langle cur, timeline, old, rlist, write, saved\_read,$
                                      $saved\_read\_ptr \rangle$

$Read(self) \triangleq \land pc[self] = \text{"Read"}$
                $\land saved\_read\_ptr' = [saved\_read\_ptr \text{ EXCEPT } ![self] = cur]$
                $\land saved\_read' = [saved\_read \text{ EXCEPT } ![self] = timeline[cur]]$
                $\land pc' = [pc \text{ EXCEPT } ![self] = \text{"Check"}]$
                $\land \text{UNCHANGED } \langle cur, hzdreclist, timeline, old, rlist, write \rangle$
```

$$\begin{aligned}
\mathit{Check}(\mathit{self}) \triangleq\ & \land\ \mathit{pc}[\mathit{self}] = \text{"Check"} \\
& \land\ \mathit{Assert}(\mathit{saved\_read}[\mathit{self}] = \mathit{timeline}[\mathit{saved\_read\_ptr}[\mathit{self}]], \\
& \qquad \text{"Failure of assertion at line 44, column 5."}) \\
& \land\ \mathit{pc}' = [\mathit{pc}\ \text{EXCEPT}\ ![\mathit{self}] = \text{"Release"}] \\
& \land\ \text{UNCHANGED}\ \langle \mathit{cur}, \mathit{hzdreclist}, \mathit{timeline}, \mathit{old}, \mathit{rlist}, \mathit{write}, \\
& \qquad \mathit{saved\_read}, \mathit{saved\_read\_ptr} \rangle
\end{aligned}$$

$$\begin{aligned}
\mathit{Release}(\mathit{self}) \triangleq\ & \land\ \mathit{pc}[\mathit{self}] = \text{"Release"} \\
& \land\ \mathit{hzdreclist}' = \mathit{hzdreclist} \setminus \{\mathit{cur}\} \\
& \land\ \mathit{pc}' = [\mathit{pc}\ \text{EXCEPT}\ ![\mathit{self}] = \text{"Done"}] \\
& \land\ \text{UNCHANGED}\ \langle \mathit{cur}, \mathit{timeline}, \mathit{old}, \mathit{rlist}, \mathit{write}, \mathit{saved\_read}, \\
& \qquad \mathit{saved\_read\_ptr} \rangle
\end{aligned}$$

$$\mathit{reader}(\mathit{self}) \triangleq \mathit{Acquire}(\mathit{self}) \lor \mathit{Read}(\mathit{self}) \lor \mathit{Check}(\mathit{self}) \lor \mathit{Release}(\mathit{self})$$

Allow infinite stuttering to prevent deadlock on termination.

$$\begin{aligned}
\mathit{Terminating} \triangleq\ & \land\ \forall\, \mathit{self} \in \mathit{ProcSet} : \mathit{pc}[\mathit{self}] = \text{"Done"} \\
& \land\ \text{UNCHANGED}\ \mathit{vars}
\end{aligned}$$

$$\begin{aligned}
\mathit{Next} \triangleq\ & \mathit{writer} \\
& \lor (\exists\, \mathit{self} \in 2 \,..\, \mathit{NumReaders} + 1 : \mathit{reader}(\mathit{self})) \\
& \lor \mathit{Terminating}
\end{aligned}$$

$$\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}}$$

$$\mathit{Termination} \triangleq \Diamond(\forall\, \mathit{self} \in \mathit{ProcSet} : \mathit{pc}[\mathit{self}] = \text{"Done"})$$

END TRANSLATION

Hazard Pointer Configuration:

```
SPECIFICATION Spec
\* Add statements after this line.
\* SPECIFICATION
\* Uncomment the previous line and provide the specification name if it's declared
\* in the specification file. Comment INIT / NEXT parameters if you use SPECIFICATION.

CONSTANTS
    NumReaders = 3
    NumWrites = 4
```

2 Reflection

2.1 Strengths and Weaknesses

TLA+ with PlusCal ended up being a great higher level modeling language that allowed me to reason closer to the impementation rather than logic. I enjoyed using the builtins for printing and support for common data structures.

I think the library system that comes by default in TLA+ leads to a more complex, descriptive set of language structures than Alloy. I think that the learning curve was actually steeper than Alloy for someone just coming from a formal methods course. I ended up leaning heavily on the translations from PlusCal to TLA to understand the underlying semantics of high level PlusCal code.

I found the concurrency model to be the perfect combination of Alloy and FSP. It allowed me to specify both structure and events in a very simple way. The disadvantage here was that the intersection of these led to really complex TLA code. TLA does not have native support for this concurrency so it ended up translating local process state to a weird, indexed global state. I think further modeling would have yielded unreadable TLA translations.

Finally, I was unable to visualize my model with TLA+. This is something that I leaned on heavily during my time with Alloy. I ended up using TLC's *print* statement which seemed like the wrong way to debug a model.

2.2 When Not to Use TLA+

I would not recommend TLA+ when the model is sufficiently complex. I found the module system in Alloy to be much simpler to understand and also to scale. State space explosion was a real issue for me.

I also think that TLA+ does not lend itself to beginners as well as advertised. Beginners will have much better success with FSP, where the model is very simple transitions with visualizations. PlusCal feels like it requires intimate knowledge of TLA to even begin, yet the claim is that PlusCal allows engineers to have a lower barrier of entry to modeling. Don't misunderstand me, I loved using PlusCal and probaly will default to using it when I need to model things in the future.

2.3 Most Important Improvement

I think that the best improvement to TLA+ would be a really robust visualization system like Alloy. Something that could be adapted by users to better show traces of the model would be extremely useful during development and also for communication to others about the model without having to show them the raw mathematics.