Lecture 5: Memory and Concurrent Access

Based on material from Chapter 5, Specifying Systems by Leslie Lamport
Today’s plan

- Review TLA+
- Specify a memory interface
- Specify linearizable memory
- Implement a linearizable cache on top of linearizable memory
- Review refinement
TLA+ review
Definition: *State*

- A *state* is an assignment of values to (*all*) variables.
- TLA+ notation: \([\text{var}_1 = \text{value}_1, \text{var}_2 = \text{value}_2, \cdots]\)
Definition: Behavior

• A behavior is a sequence of states
• Notation: \( state_1 \rightarrow state_2 \rightarrow state_3 \rightarrow \cdots \)
• Example: \([hr = 11] \rightarrow [hr = 12] \rightarrow [hr = 1]\)
Definition: *Step*

- A *step* consists of two consecutive states in a behavior
- aka *transition*
- Notation: $state_1 \rightarrow state_2$
- Example: $[hr = 3] \rightarrow [hr = 4]$
Definition: *Specification*

- A *specification* is a set of all possible behaviors
- Consists of at least two parts
  1. Set of all possible *initial states*
  2. A “next-state” relation that describes the ways a state may change in a step
    - i.e., the set of all possible pairs of states
- May also contain a liveness condition and some theorems
Set of Initial States

• Example: \( HC_{ini} \triangleq hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\} \)

• A set of states can often be succinctly described by a predicate
  • Example: \( HC_{ini} \triangleq hr \in \mathbb{N} \land 1 \leq hr \land hr \leq 12 \)

```
Definition: *Action*

- An *action* is a predicate over a pair of states in a step.
- Example: $HC\text{nxt} \triangleq hr' = hr \mod 12 + 1$
- $hr'$ is the value of $hr$ in the new state; $hr$ is the value in the old state.
Definition: *Stuttering steps*

- A stuttering step keeps (certain) state variable unchanged
- Example:
  
  \[ [hr' = hr \mod 12 + 1]_{hr} \triangleq (hr' = hr \mod 12 + 1) \lor (hr' = hr) \]
Definition: *State Function/Predicate*

- A *state function* is a first-order logic expression
- A *state predicate* is a Boolean state function
Definition: *Temporal Formula*

- A *temporal formula* $F$ assigns a Boolean value to a behavior $\sigma$
- $\sigma \models F$ means that $F$ holds over $\sigma$
- If $P$ is a state predicate, then $\sigma \models P$ means that $P$ holds over the first state in $\sigma$
- If $A$ is an action, then $\sigma \models A$ means that $A$ holds over the first two states in $\sigma$
- If $A$ is an action, then $\sigma \models [A]_\nu$ means that the first step in $\sigma$ is an $A$ step or a stuttering step with respect to $\nu$
Always

• $\sigma \models \Box F$ means that $F$ holds over every suffix of $\sigma$

• More formally
  • Let $\sigma^{+n}$ be $\sigma$ with the first $n$ states removed
  • Then $\sigma \models \Box F \iff \forall n \in \mathbb{N} : \sigma^{+n} \models F$
Example specification: hardware clock

Module HourClock

• VARIABLE $hr$
• $HC_{ini} \triangleq hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 \}$
• $HC_{nxt} \triangleq hr' = hr \mod 12 + 1$
• $HC \triangleq HC_{ini} \land \Box[HC_{nxt}]_{hr}$
Definition: *Theorem*

- A *theorem* is a temporal formula that holds over every behavior of the specification.
- Example: \( HC \Rightarrow \Box hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\} \)
  - That is, \( HC \Rightarrow \Box HC_{ini} \)
Definition: *Invariant*

- If $S$ is a specification and $I$ is a predicate and $S \Rightarrow \Box I$ is a theorem then we call $I$ an *invariant* of $S$. 
Modeling Shared Memory
(naïve) attempt: function [Address $\mapsto$ Value]

- CONSTANTS $Adr, Val$
- VARIABLE $mem$
- TypeInvariant $\triangleq mem \in [Adr \rightarrow Val]$
- Read($a$) $\triangleq mem[a]$
- Write($a, v$) $\triangleq mem' = [mem \ \text{EXCEPT} \! [a] = v]$
(naïve) attempt 1: function [Address $\mapsto$ Value]

• CONSTANTS $Adr, Val$
• VARIABLE $mem$
• TypeInvariant $\triangleq mem \in [Adr \to Val]$
• Read$(a) \triangleq mem[a]$
• Write$(a, v) \triangleq mem' = [mem \text{ EXCEPT } ! [a] = v]$

Ignores how processes interact with memory
Ignores memory coherence properties
A Memory System

An abstract memory interface
MODULE MemoryInterface

VARIABLE memInt

CONSTANTS Send(−, −, −, −), A Send(p, d, memInt, memInt') step represents processor p
sending value d to the memory.

Reply(−, −, −, −), A Reply(p, d, memInt, memInt') step represents the memory
sending value d to processor p.

InitMemInt, The set of possible initial values of memInt.
Proc, The set of processor identifiers.
Adr, The set of memory addresses.
Val The set of memory values.

ASSUME ∀ p, d, miOld, miNew : ∀ Send(p, d, miOld, miNew) ∈ BOOLEAN
∧ Reply(p, d, miOld, miNew) ∈ BOOLEAN
MODULE MemoryInterface

VARIABLE memInt

CONSTANTS Send(_, _, _, _,_), A $Send(p, d, \text{memInt}, \text{memInt}')$ step represents processor $p$ sending value $d$ to the memory.

Reply(_, _, _, _), A $Reply(p, d, \text{memInt}, \text{memInt}')$ step represents the memory sending value $d$ to processor $p$.

InitMemInt, The set of possible initial values of memInt.

Proc, The set of processor identifiers.

Adr, The set of memory addresses.

Val The set of memory values.

ASSUME $\forall p, d, \text{miOld}, \text{miNew} : \land \ Send(p, d, \text{miOld}, \text{miNew}) \in \text{BOOLEAN}$

$\land \ Reply(p, d, \text{miOld}, \text{miNew}) \in \text{BOOLEAN}$
MODULE MemoryInterface

VARIABLE memInt

CONSTANTS Send(\_\_\_\_\_), Reply(\_\_\_\_\_), InitMemInt, Proc, Adr, Val

A Send(p, d, memInt, memInt') step represents processor p sending value d to the memory.
A Reply(p, d, memInt, memInt') step represents the memory sending value d to processor p.

InitMemInt, The set of possible initial values of memInt.
Proc, The set of processor identifiers.
Adr, The set of memory addresses.
Val The set of memory values.

ASSUME ∀ p, d, miOld, miNew : ∀ Send(p, d, miOld, miNew) ∈ BOOLEAN
∧ Reply(p, d, miOld, miNew) ∈ BOOLEAN

MReq \doteq [op : \{“Rd”\}, adr : Adr] ∪ [op : \{“Wr”\}, adr : Adr, val : Val]
The set of all requests; a read specifies an address, a write specifies an address and a value.
A Memory System

Processor → Mreq → memInt

Val for reads

NoVal for writes

Processor → MEMORY
MODULE MemoryInterface

VARIABLE memInt

CONSTANTS Send( , , , ), A Send(p, d, memInt, memInt') step represents processor p sending value d to the memory.

Reply( , , , ), A Reply(p, d, memInt, memInt') step represents the memory sending value d to processor p.

InitMemInt, The set of possible initial values of memInt.
Proc, The set of processor identifiers.
Adr, The set of memory addresses.
Val The set of memory values.

ASSUME ∀ p, d, miOld, miNew : ∧ Send(p, d, miOld, miNew) ∈ BOOLEAN
∧ Reply(p, d, miOld, miNew) ∈ BOOLEAN

MReq ⊆ [op : {“Rd”}, adr : Adr] ∪ [op : {“Wr”}, adr : Adr, val : Val]
The set of all requests; a read specifies an address, a write specifies an address and a value.

NoVal ⊆ CHOOSE v : v ∉ Val An arbitrary value not in Val.
A Linearizable Memory System
Linearizability [Herlihy & Wing 1990]
MODULE InternalMemory

EXTENDS MemoryInterface

VARIABLES mem, ctl, buf

\[ IInit \triangleq \text{The initial predicate} \]
\[ \land \ mem \in [Adr \rightarrow Val] \]
\[ \land \ ctl = [p \in Proc \mapsto \text{"rdy"}] \]
\[ \land \ buf = [p \in Proc \mapsto \text{NoVal}] \]
\[ \land \ memInt \in \text{InitMemInt} \]

Initially, memory locations have any values in Val, each processor is ready to issue requests, each \( buf[p] \) is arbitrarily initialized to NoVal, and \( memInt \) is any element of \( \text{InitMemInt} \).

\[ \text{TypeInvariant} \triangleq \text{The type-correctness invariant.} \]
\[ \land \ mem \in [Adr \rightarrow Val] \]
\[ \land \ ctl \in [Proc \rightarrow \{\text{"rdy"}, \text{"busy"}, \text{"done"}\}] \]
\[ \land \ buf \in [Proc \rightarrow MReq \cup Val \cup \{\text{NoVal}\}] \]

\( mem \) is a function from \( Adr \) to \( Val \).
\( ctl[p] \) equals \( \text{"rdy"}, \text{"busy"}, \) or \( \text{"done"} \).
\( buf[p] \) is a request or a response.
Behaviors

\[ req \equiv [\text{op} \leftrightarrow "Wr", \text{adr} \leftrightarrow a, \text{val} \leftrightarrow v] \]

\[
\begin{align*}
\text{ctl}[p] &= "rdy" & \text{buf}[p] &= \ldots & \text{mem}[a] &= \ldots \\
\text{mem}[a] &= v
\end{align*}
\]

\[ \text{ctl}[p] = "busy", \text{buf}[p] = \text{req} \]

\[ \begin{align*}
\text{ctl}[p] &= "done" & \text{buf}[p] &= \text{NoVal} & \text{mem}[a] &= v \\
\text{buf}[p] &= \text{Req}(p) & \text{mem}[a] &= \ldots
\end{align*} \]

\[ \text{ctl}[p] = "rdy" \]

\[ \text{buf}[p] = \text{NoVal} \]

\[ \text{mem}[a] = v \]

\[ req \equiv [\text{op} \leftrightarrow "Rd", \text{adr} \leftrightarrow a] \]

\[
\begin{align*}
\text{ctl}[p] &= "rdy" & \text{buf}[p] &= \ldots & \text{mem}[a] &= \ldots \\
\text{mem}[a] &= v
\end{align*}
\]

\[ \text{ctl}[p] = "busy", \text{buf}[p] = \text{req} \]

\[ \begin{align*}
\text{ctl}[p] &= "done" & \text{buf}[p] &= v & \text{mem}[a] &= \ldots \\
\text{buf}[p] &= \text{Req}(p) & \text{mem}[a] &= \ldots
\end{align*} \]

\[ \text{ctl}[p] = "rdy" \]

\[ \text{buf}[p] = v \]

\[ \text{mem}[a] = \ldots \]
\( \text{Req}(p) \triangleq \) Processor \( p \) issues a request.

\( \land \ ctl[p] = \text{“rdy”} \)  Enabled iff \( p \) is ready to issue a request.

\( \land \ \exists \ req \in M\text{Req} : \)  For some request \( req \):

\( \land \ \text{Send}(p, req, \text{memInt}, \text{memInt'}) \)  Send \( req \) on the interface.

\( \land \ \text{buf}' = [\text{buf EXCEPT ![p] = req}] \)  Set \( \text{buf}[p] \) to the request.

\( \land \ \text{ctl}' = [\text{ctl EXCEPT ![p] = “busy”}] \)  Set \( \text{ctl}[p] \) to “busy”.

\( \land \ \text{UNCHANGED mem} \)
$Do(p) \triangleq$ Perform $p$'s request to memory.

$\land \ qtl[p] = \text{"busy"}$ Enabled iff $p$'s request is pending.

$\land \ mem' = \text{IF} \ buf[p].op = \text{"Wr"}$
$\hspace{1cm}$ THEN $[mem \ \text{EXCEPT}
\hspace{1cm}!\{buf[p].adr\} = buf[p].val]\hspace{1cm}$ Write to memory on a 
$\hspace{1cm}$ "Wr" request.

$\hspace{1cm}$ ELSE $mem$ Leave $mem$ unchanged on a "Rd" request.

$\land \ buf' = [buf \ \text{EXCEPT} \ !\{p\} = \text{IF} \ buf[p].op = \text{"Wr"}$
$\hspace{1cm}$ THEN $NoVal$
$\hspace{1cm}$ ELSE $mem[buf[p].adr]\hspace{1cm}$ Set $buf[p]$ to the response:
$\hspace{1cm}$ $NoVal$ for a write;

$\land \ qtl' = [ctl \ \text{EXCEPT} \ !\{p\} = \text{"done"}]$ 

$\land \ \text{UNCHANGED} \ memInt$

\[
\begin{bmatrix}
ctl[p] &= "busy" \\
buf[p] &= req \\
mem[a] &= \ldots
\end{bmatrix}
\xrightarrow{Do(p)}
\begin{bmatrix}
ctl[p] &= "done" \\
buf[p] &= NoVal \\
mem[a] &= v
\end{bmatrix}
\xrightarrow{Rsp(p)}
\]
\( Rsp(p) \triangleq \) Return the response to \( p \)'s request.
\[ \land \ ctl[p] = \text{“done”} \]
\[ \land \ Reply(p, \ buf[p], \ memInt, \ memInt') \]
\[ \land \ ctl' = [ctl \ \text{EXCEPT} \ !p = \text{“rdy”}] \]
\[ \land \ \text{UNCHANGED} \ <mem, \ buf> \]

Enabled iff req. is done but resp. not sent.
Send the response on the interface.
Set \( ctl[p] \) to \text{“rdy”}.

\[
\begin{align*}
ctl[p] &= \text{"done"} \\
buf[p] &= \text{NoVal} \\
mem[a] &= v
\end{align*}
\]
\[ \xrightarrow{Rsp(p)} \]
\[
\begin{align*}
ctl[p] &= \text{"rdy"} \\
buf[p] &= \text{NoVal} \\
mem[a] &= v
\end{align*}
\]
MODULE InternalMemory

EXTENDS MemoryInterface

VARIABLES mem, ctl, buf

\[ IInit \triangleq \text{The initial predicate} \]
\[ \land \text{mem} \in [\text{Adr} \rightarrow \text{Val}] \]
\[ \land \text{ctl} = \{ p \in \text{Proc} \mapsto \text{"rdy"} \} \]
\[ \land \text{buf} = \{ p \in \text{Proc} \mapsto \text{NoVal} \} \]
\[ \land \text{memInt} \in \text{InitMemInt} \]

Initially, memory locations have any values in Val, each processor is ready to issue requests, each buf[p] is arbitrarily initialized to NoVal, and memInt is any element of InitMemInt.

\[ INext \triangleq \exists p \in \text{Proc} : \text{Req}(p) \lor \text{Do}(p) \lor \text{Rsp}(p) \]
The next-state action.

\[ ISpec \triangleq IInit \land \Box[INext]_{\langle \text{memInt}, \text{mem}, \text{ctl}, \text{buf} \rangle} \]
The specification.

THEOREM ISpec \Rightarrow \Box TypeInvariant
Note, this is a “specification” describing behaviors of linearizable memory more than an “implementation” (mapping onto physical hardware). Of course, both can be described by TLA+ formulas.
Implementing a Write-Through Cache

Implements the linearizable memory interface (on top of another)
EXTENDS Naturals, Sequences, MemoryInterface

VARIABLES wmem, ctl, buf, cache, memQ

CONSTANT QLen

ASSUME \((QLen \in \text{Nat}) \wedge (QLen > 0)\)

\[ M \triangleq \text{INSTANCE InternalMemory with } \text{mem} \leftarrow \text{wmem} \]

\(\text{Init} \triangleq \) The initial predicate

\(\wedge M \not\triangleright \text{Init} \) wmem, buf, and ctl are initialized as in the internal memory spec.

\(\wedge \text{cache} = \) All caches are initially empty \((\text{cache}[p][a] = \text{NoVal} \text{ for all } p, a)\).

\(\left[ p \in \text{Proc} \mapsto [a \in \text{Adr} \mapsto \text{NoVal}] \right] \)

\(\wedge \text{memQ} = \langle \rangle \) The queue memQ is initially empty.

\(\text{TypeInvariant} \triangleq \) The type invariant.

\(\wedge \text{wmem} \in [\text{Adr} \rightarrow \text{Val}]\)

\(\wedge \text{ctl} \in [\text{Proc} \rightarrow \{\text{"rdy"}, \text{"busy"}, \text{"waiting"}, \text{"done"}\}]\)

\(\wedge \text{buf} \in [\text{Proc} \rightarrow \text{MReq} \cup \text{Val} \cup \{\text{NoVal}\}]\)

\(\wedge \text{cache} \in [\text{Proc} \rightarrow [\text{Adr} \rightarrow \text{Val} \cup \{\text{NoVal}\}]\]

\(\wedge \text{memQ} \in \text{Seq} (\text{Proc} \times \text{MReq}) \) memQ is a sequence of \(\langle \text{proc.}, \text{request} \rangle\) pairs.
Note

In TLA+

• Sequences and tuples are functions \([[1 \ldots ] \rightarrow \text{values}]]
• Recall: records are functions \([\text{String} \rightarrow \text{values}]\)
EXTENDS Naturals, Sequences, MemoryInterface

VARIABLES \( \text{wmem, ctl, buf, cache, memQ} \)

CONSTANT \( QLen \)

ASSUME \((QLen \in \text{Nat}) \land (QLen > 0)\)

\[ M \triangleq \text{INSTANCE InternalMemory WITH } \text{mem} \leftarrow \text{wmem} \]

\[ \text{Init} \triangleq \text{The initial predicate} \]
\[ \land M \not\models \text{Init} \]
\[ \land \text{cache} = \text{All caches are initially empty (cache}[p][a] = \text{NoVal} \text{ for all } p, a).} \]
\[ [p \in \text{Proc} \leftrightarrow (a \in \text{Adr} \rightarrow \text{NoVal})] \]
\[ \land \text{memQ} = \langle \rangle \text{ The queue memQ is initially empty.} \]

\[ \text{TypeInvariant} \triangleq \text{The type invariant.} \]
\[ \land \text{wmem} \in [\text{Adr} \rightarrow \text{Val}] \]
\[ \land \text{ctl} \in [\text{Proc} \rightarrow \{\text{"rdy", "busy", "waiting", "done"}\}] \]
\[ \land \text{buf} \in [\text{Proc} \rightarrow M\text{Req} \cup \text{Val} \cup \{\text{NoVal}\}] \]
\[ \land \text{cache} \in [\text{Proc} \rightarrow [\text{Adr} \rightarrow \text{Val} \cup \{\text{NoVal}\}]] \]
\[ \land \text{memQ} \in \text{Seq}(\text{Proc} \times M\text{Req}) \text{ memQ is a sequence of (proc., request) pairs.} \]
Cache Coherence

\[ Coherence \triangleq \forall p, q \in \text{Proc}, a \in \text{Adr} : \]
\[
(\text{NoVal} \notin \{\text{cache}[p][a], \text{cache}[q][a]\}) \Rightarrow (\text{cache}[p][a] = \text{cache}[q][a])
\]

Asserts that if two processors’ caches both have copies of an address, then those copies have equal values.
External Interface is the same

\[
\begin{align*}
\text{Req}(p) & \triangleq \text{Processor } p \text{ issues a request.} \\
M!\text{Req}(p) & \land \text{UNCHANGED } \langle \text{cache, memQ} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{Rsp}(p) & \triangleq \text{The system issues a response to processor } p. \\
M!\text{Rsp}(p) & \land \text{UNCHANGED } \langle \text{cache, memQ} \rangle
\end{align*}
\]

\[
\begin{bmatrix}
\text{ctl}[p] & = & "\text{rdy}" \\
\text{buf}[p] & = & \ldots
\end{bmatrix}_{\text{Req}(p)} \Rightarrow ???????? \Rightarrow ???????? \Rightarrow \begin{bmatrix}
\text{ctl}[p] & = & "\text{rdy}" \\
\text{buf}[p] & = & \ldots
\end{bmatrix}
\]
\( \text{DoWr}(p) \triangleq \) Write to \( p \)'s cache, update other caches, and enqueue memory update.

\( \text{LET } r \triangleq \text{buf}[p] \) Processor \( p \)'s request.

\( \text{IN } \wedge (\text{ctl}[p] = \text{"busy"}) \wedge (r.op = \text{"Wr"}) \wedge \text{Len}(\text{memQ}) < \text{QLen} \)

\( \wedge \text{cache}' = \) Update \( p \)'s cache and any other cache that has a copy.

\[ q \in \text{Proc} \mapsto \text{IF } (p = q) \lor (\text{cache}[q][r.adr] \neq \text{NoVal}) \]
\[ \text{THEN } [\text{cache}[q] \text{ EXCEPT } ![r.adr] = r.val] \]
\[ \text{ELSE } \text{cache}[q] \]

\( \wedge \text{memQ}' = \text{Append}(\text{memQ}, \langle p, r \rangle) \)

\( \wedge \text{buf}' = [\text{buf} \text{ EXCEPT } ![p] = \text{NoVal}] \)

\( \wedge \text{ctl}' = [\text{ctl} \text{ EXCEPT } ![p] = \text{"done"}] \)

\( \wedge \text{UNCHANGED } \langle \text{memInt}, \text{wmem} \rangle \)

Enabled if write request pending and \( \text{memQ} \) is not full.

Enqueue write at tail of \( \text{memQ} \).

Generate response.

Set \( \text{ctl} \) to indicate request is done.
\( \text{MemQWr} \triangleq \text{Perform write at head of memQ to memory.} \)

\[
\begin{align*}
\text{LET } r & \triangleq \text{Head}(\text{memQ})[2] \quad \text{The request at the head of memQ.} \\
\text{IN} & \quad \land (\text{memQ} \neq \langle \rangle) \land (r.\text{op} = \text{"Wr"}) \\
& \quad \land \text{wmem}' = \\
& \quad \quad \quad [\text{wmem EXCEPT } ![r.\text{adr}] = r.\text{val}] \\
& \quad \land \text{memQ}' = \text{Tail}(\text{memQ}) \\
& \quad \land \text{UNCHANGED } \langle \text{memInt, buf, ctl, cache} \rangle
\end{align*}
\]

Enabled if \( \text{Head}(\text{memQ}) \) a write. Perform the write to memory. Remove the write from \( \text{memQ} \).
RdMiss$(p) \triangleq$ Enqueue a request to write value from memory to $p$’s cache.

\[ \land (ctl[p] = \text{"busy"}) \land (buf[p].op = \text{"Rd"}) \land cache[p][buf[p].adr] = \text{NoVal} \land Len(memQ) < QLen \land memQ' = \text{Append}(memQ, \langle p, buf[p] \rangle) \land ctl' = [ctl \text{ EXCEPT } ![p] = \text{"waiting"}] \land \text{UNCHANGED } \langle \text{memInt, wmem, buf, cache} \rangle \]

Enabled on a read request when the address is not in $p$’s cache and $memQ$ is not full. Append $\langle p, \text{request} \rangle$ to $memQ$. Set $ctl[p]$ to “waiting”.

DoRd$(p) \triangleq$ Perform a read by $p$ of a value in its cache.

\[ \land ctl[p] \in \{\text{"busy"}, \text{"waiting"}\} \land buf[p].op = \text{"Rd"} \land cache[p][buf[p].adr] \neq \text{NoVal} \land buf' = [buf \text{ EXCEPT } ![p] = cache[p][buf[p].adr]] \land ctl' = [ctl \text{ EXCEPT } ![p] = \text{"done"}] \land \text{UNCHANGED } \langle \text{memInt, wmem, cache, memQ} \rangle \]

Enabled if a read request is pending and address is in cache. Get result from cache. Set $ctl[p]$ to “done”.
$vmem \triangleq$ The value $wmem$ will have after all the writes in $memQ$ are performed.

Let $f[i \in 0..\text{Len}(memQ)]$ 

If $i = 0$ then $wmem$

Else if $memQ[i][2].op = \text{"Rd"}$

Then $f[i - 1]$

Else $[f[i - 1] \text{ EXCEPT } ![memQ[i][2].adr] = memQ[i][2].val]$
Completing the spec

\[ Evict(p, a) \triangleq\text{ Remove address } a \text{ from } p'\text{'}s \text{ cache.} \]
\[ \land (ctl[p] = \text{“waiting”}) \Rightarrow (buf[p].adr \neq a) \]
\[ \land \text{cache}' = [\text{cache EXCEPT } ![p][a] = \text{NoVal}] \]
\[ \land \text{UNCHANGED } \langle \text{memInt, wmem, buf, ctl, memQ} \rangle \]

\[ \text{Next } \triangleq \forall \exists p \in \text{Proc} : \forall \text{Req}(p) \lor \text{Rsp}(p) \]
\[ \lor \text{RdMiss}(p) \lor \text{DoRd}(p) \lor \text{DoWr}(p) \]
\[ \lor \exists a \in \text{Adr} : \text{Evict}(p, a) \]
\[ \lor \text{MemQWr} \lor \text{MemQRd} \]

\[ \text{Spec } \triangleq \text{Init} \land \Box[\text{Next}] \langle \text{memInt, wmem, buf, ctl, cache, memQ} \rangle \]
Theorems

THEOREM \( Spec \Rightarrow \square TypeInvariant \)
THEOREM \( Spec \Rightarrow \square Coherence \)

More like lemmas

\[ LM \triangleq \text{INSTANCE Memory} \]
THEOREM \( Spec \Rightarrow LM!Spec \)
Inductive Invariants

**THEOREM** \( TypeInvariant \land \text{Next} \Rightarrow TypeInvariant' \)

\( TypeInvariant \) is an invariant of the next-state action

Thus, if \( TypeInvariant \) holds over initial states, by induction it holds over all states
Coherence is not an inductive invariant

• Consider a state in which:
  • $cache[p1][a] = 1$
  • $\forall \langle q, b \rangle: cache[q][b] = NoVal$
  • $wmem[a] = 2$
  • $memQ = \langle \langle p2, [op \mapsto "Rd", adr \mapsto a] \rangle \rangle$

• Now take the $MemQRd$ step:
  • $cache[p1][a] = 1$
  • $cache[p2][a] = 2$

Need to prove an inductive invariant that implies Coherence

Suggestions?
A proposed stronger invariant

- Recall that function $vmem$ represents current state of memory
- Inductive Invariant:
  $\forall p \in Proc, a \in Adr: (cache[p][a] = NoVal) \lor (cache[p][a] = vmem[a])$
- Implies Coherence
Proving $Spec \Rightarrow LM!Spec$

By definition of $LM!Spec$, we need to prove

\[ \text{THEOREM } Spec \Rightarrow \exists \text{ mem, ctl, buf : } LM!\text{Inner}(\text{mem, ctl, buf})!ISpec \]

Which means we have to find "witnesses" for $\text{mem, ctl and buf}$: this is called a refinement mapping

Any guesses?
Proving $Spec \Rightarrow LM!Spec$

By definition of $LM!Spec$, we need to prove

$$\text{THEOREM } Spec \Rightarrow \exists \text{ mem, ctl, buf : } LM!Inner(\text{mem, ctl, buf})!ISpec$$

Which means we have to find “witnesses” for $\text{mem, ctl}$ and $\text{buf}$: this is called a refinement mapping:

$$\begin{align*}
\text{omem} & \triangleq \text{vmem} \\
\text{octl} & \triangleq [p \in \text{Proc} \mapsto \text{IF } \text{ctl}[p] = \text{“waiting” } \text{THEN } \text{“busy” } \text{ELSE } \text{ctl}[p]] \\
\text{obuf} & \triangleq \text{buf}
\end{align*}$$
MODULE InternalMemory

EXTENDS MemoryInterface

VARIABLES mem, ctl, buf

\[ IInit \triangleq \begin{align*}
\land \ mem \in [Adr \to Val] \\
\land \ ctl = [p \in Proc \mapsto \text{"rdy"}] \\
\land \ buf = [p \in Proc \mapsto \text{NoVal}] \\
\land \ memInt \in InitMemInt
\end{align*} \]

Initially, memory locations have any values in Val, each processor is ready to issue requests, each \( buf[p] \) is arbitrarily initialized to \text{NoVal}, and \( memInt \) is any element of \( \text{InitMemInt} \).

\[ INext \triangleq \exists p \in Proc : Req(p) \lor Do(p) \lor Rsp(p) \]

The next-state action.

\[ ISpec \triangleq IInit \land \square [INext]_{\langle memInt, mem, ctl, buf \rangle} \]

The specification.

THEOREM \( ISpec \Rightarrow \square \text{TypeInvariant} \)
Proving refinement

• If $F$ is a formula of module $\textit{InternalMemory}$ (the high-level spec), let
  
  $\overline{F} \equiv \textit{LM! Inner}(\textit{omem}, \textit{octl}, \textit{obuf})$

  • That is: $F$ with $\textit{omem}$, $\textit{octl}$, and $\textit{obuf}$ substituted for $\textit{mem}$, $\textit{ctl}$, and $\textit{buf}$

• Then we need to prove that $\textit{Spec} \Rightarrow \overline{\textit{ISpec}}$

• Replacing definitions, we need to prove:

\[
\text{Init} \land \Box[\text{Next}] \langle \textit{memInt}, \textit{wmem}, \textit{buf}, \textit{ctl}, \textit{cache}, \textit{memQ} \rangle
\Rightarrow \overline{\text{Init}} \land \Box[\overline{\text{INext}}] \langle \textit{memInt}, \overline{\textit{mem}}, \overline{\textit{ctl}}, \overline{\textit{buf}} \rangle
\]

• Find an invariant $\textit{Inv}$:

\[
\text{Init} \Rightarrow \overline{\text{Init}}
\]

\[
\text{Inv} \land \text{Next} \Rightarrow \lor \overline{\text{INext}}
\]

\[
\lor \text{UNCHANGED} \langle \textit{memInt}, \overline{\textit{mem}}, \overline{\textit{ctl}}, \overline{\textit{buf}} \rangle
\]


Show every step of $\textit{WriteThroughCache}$ is a step of $\textit{InternalMemory}$ or a stuttering step of $\textit{InternalMemory}$
About memory

• Real memory is not linearizable
  • Linearizability is not strong enough for modern processors that submit multiple requests to memory
    • If a processor submits a write and, before completion, a read to the same address, linearizability would allow the second operation to be ordered before the first
  • Linearizability is too strong for concurrent processing
    • If p1 submits operation o1 and p2 submits operation o2 and o1 completes before o2, we do not need to require that o1 is ordered before o2 (use locks if you need that)

• Sequential Consistency is more realistic and easier to implement
  • Serializability: result of execution same as some total order of operations
  • Local ordering: operations of a process ordered in submission order

• See Figure 11.7 in *Specifying Systems*
Final words

• We use TLA+ to model a system. You get to choose a level of abstraction. Choose it too high and you won’t reveal problems. Choose it too low and you get stuck in the weeds.

• Choosing the level of abstraction involves choosing what constitutes (atomic) steps: grain of atomicity

• Also involves how accurately to model the state (data structures). Consider where you are trying to reveal problems.