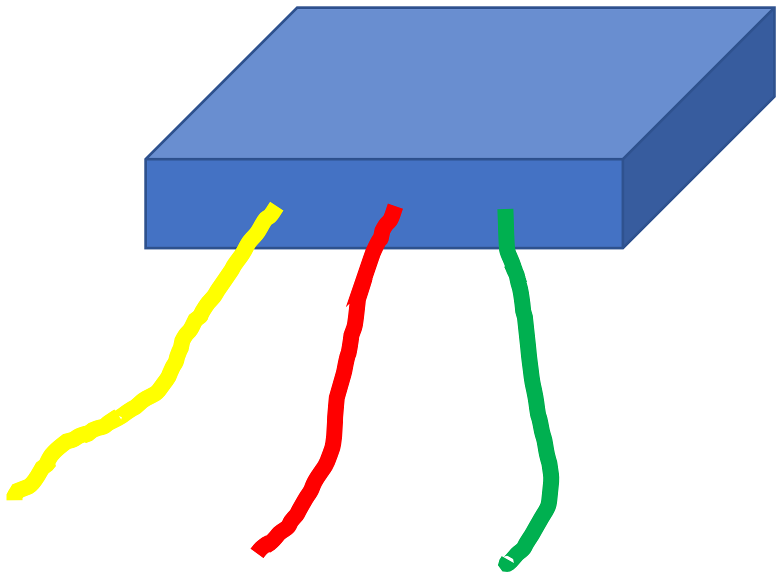
# Lecture 4: Refinement

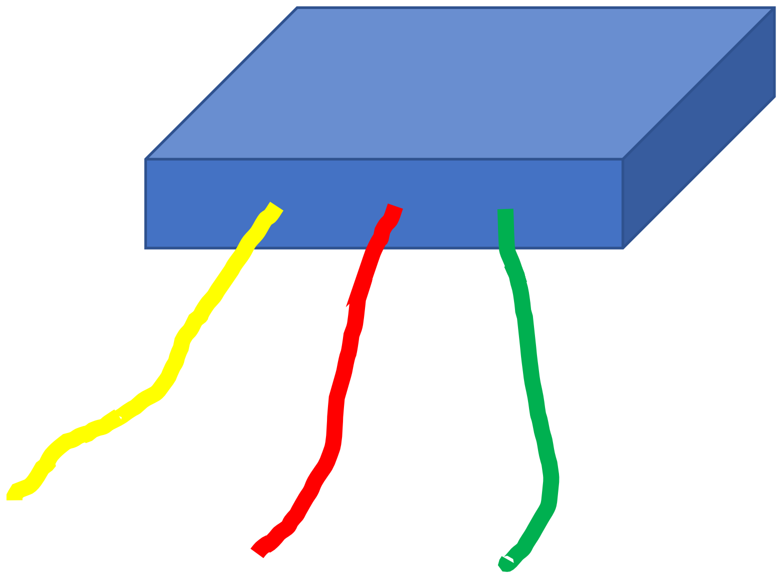
Based on material from Section 10.8, Specifying Systems by Leslie Lamport

# You ask for:



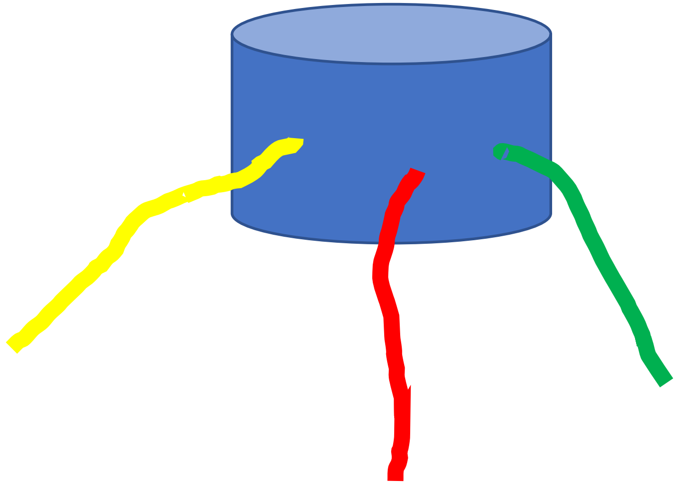
Specification

You ask for:



Specification

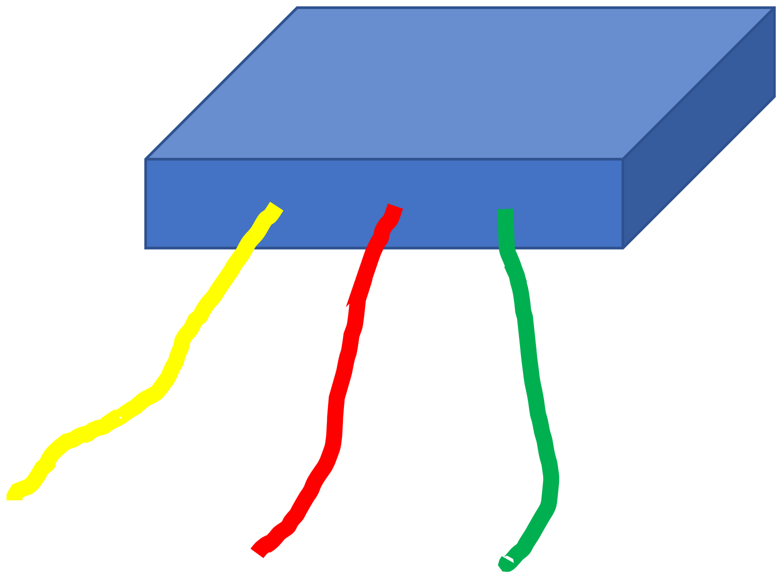
You get:



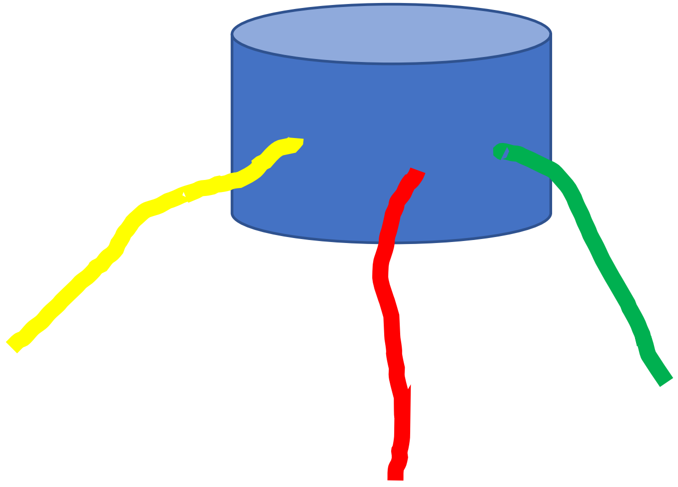
Implementation

You ask for:

You get:

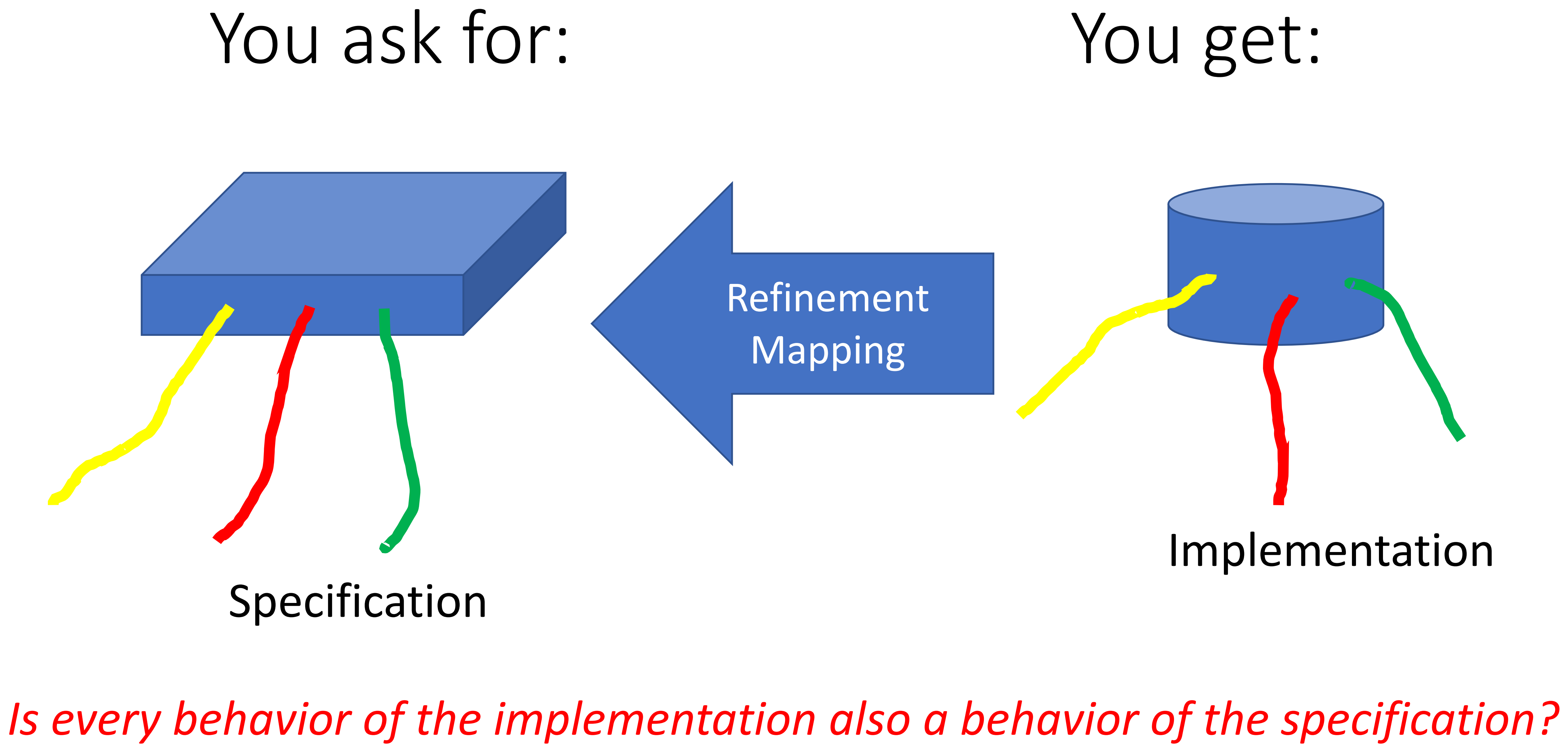


Specification



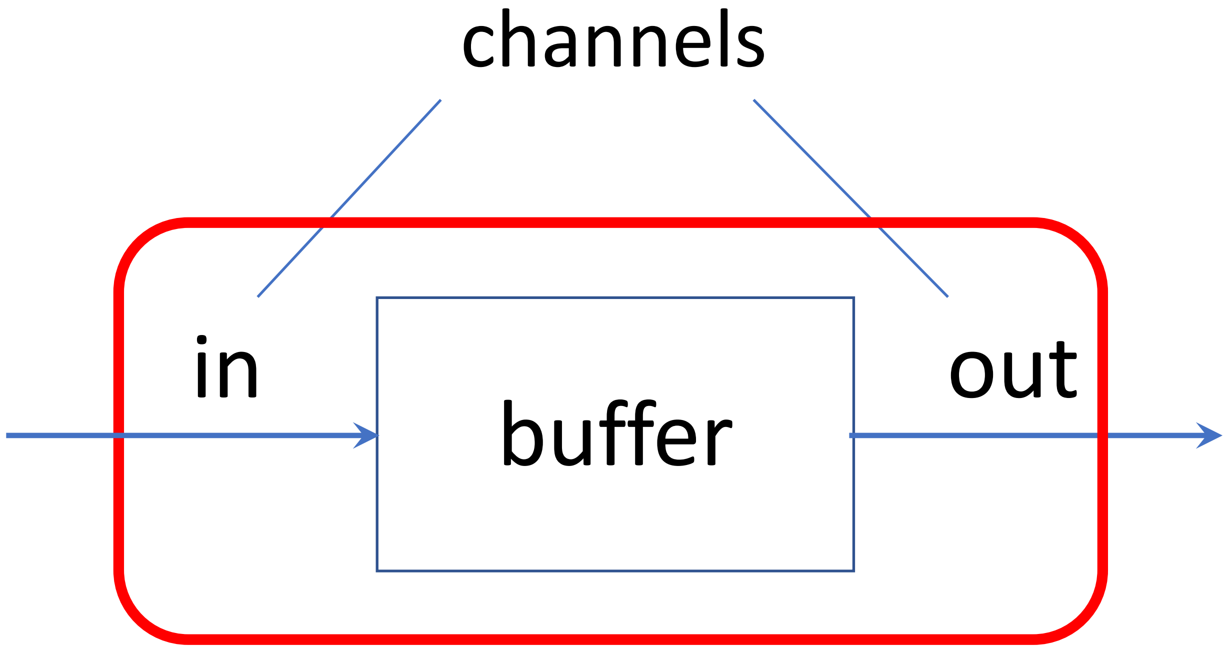
Implementation

*Is every behavior of the implementation also a behavior of the specification?*


You ask for:
You get:
Refinement
Mapping
Specification
Implementation
Is every behavior of the implementation also a behavior of the specification?

# External/internal variables of a state

- A specification has certain *external variables* that can be observed and/or manipulated
- It may also have *internal variables* that are used to describe behaviors but that cannot be observed
- Example: FIFO
  - External variables: in, out
  - Internal variable: buffer



# *Externally visible* vs *complete* behavior

A system may exhibit externally visible behavior

$$e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_4 \rightarrow \dots$$

if there exists a complete behavior

$$(e_1, y_1) \rightarrow (e_2, y_2) \rightarrow (e_3, y_3) \rightarrow (e_4, y_4) \rightarrow$$

that is allowed by the specification

Here $e_i$ is some externally visible state (for example, in and out channels) and $y_i$ is internal state (for example, the buffer)

# Stuttering Steps

A specification should allow changes to the internal state that does not change the externally visible state.

For example:

$$(e_1, y_1) \rightarrow (e_2, y_2) \rightarrow (e_2, y_2') \rightarrow (e_3, y_3) \rightarrow (e_4, y_4) \rightarrow$$

leads to external behavior

$$e_1 \rightarrow e_2 \rightarrow e_2 \rightarrow e_3 \rightarrow e_4 \rightarrow \ldots$$

which should be identical to

$$e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_4 \rightarrow \ldots$$

# Proving that an implementation meets the specification

- First note that an implementation is just a specification
- We call the implementation the “lower-level” specification

We need to prove that if an implementation allows the complete behavior

$$(e_1, z_1) \rightarrow (e_2, z_2) \rightarrow (e_3, z_3) \rightarrow (e_4, z_4) \rightarrow$$

then there exists a complete behavior

$$(e_1, y_1) \rightarrow (e_2, y_2) \rightarrow (e_3, y_3) \rightarrow (e_4, y_4) \rightarrow$$

allowed by the specification

A mapping from low-level complete behaviors to high-level complete behaviors is called a “refinement mapping”

Note, there may be multiple possible refinement mappings---you only need to show one

# Recall: Module HourClock

$$\text{MODULE } \mathit{HourClock}$$

$\text{EXTENDS } \mathit{Naturals}$

$\text{VARIABLE } hr$

$HCini \triangleq hr \in (1 \mathrel{..} 12)$

$HCnxt \triangleq hr' = \text{IF } hr \neq 12 \text{ THEN } hr + 1 \text{ ELSE } 1$

$HC \triangleq HCini \land \Box[HCnxt]_{hr}$

$\text{THEOREM } HC \Rightarrow \Box HCini$

# Implementation

- Suppose we wanted to replace *hr* by a 4-bit binary value
- We need a way to represent *n*-bit binary values
- We also need a function of *n*-bit binary values to numbers

# Functions in TLA+

- A function $f$ has a domain, written DOMAIN $f$
- $f$ assigns to each $x \in \text{DOMAIN}\, f$ a value $f[x]$
  - TLA+ uses array notation (square brackets) rather than parentheses
- $f \equiv g$ iff

$$\text{DOMAIN}\ f = \text{DOMAIN}\ g \land \forall x \in \text{DOMAIN}\ f\colon\ f[x] = g[x]$$

- The range of f is $\{\ f[x] \mid x \in \text{DOMAIN}\ f\ \}$
- $[S \rightarrow T]$ is defined to be the set of functions whose domain is $S$ and whose range is a subset of $T$

# Function values

- $[x \in S \mapsto e]$ is defined to be the function $f$ with domain $S$ such that
  $\forall x \in S\colon\ f[x] = e$ // $x$ is a free variable in $e$
- For example
  - $succ \ \triangleq\ [n \in 1..12 \mapsto \text{IF}\, n = 12 \text{ THEN } 1 \text{ ELSE}\, n + 1]$
  - $prod \ \triangleq\ [x \in Real, y \in Real \mapsto x * y]$
  - $double \ \triangleq\ [x \in Real \mapsto prod[x][2]]$
- Similar to lambda expressions

# Cool: Records are functions

$[\ val \longmapsto 42, rdy \longmapsto 1, ack \longmapsto 0]$ is equivalent to

$[\ x \in \{\text{"val"}, \text{"rdy"}, \text{"ack"}\} \longmapsto$

IF $x =$ "val" THEN 42

ELSE IF $x =$ "rdy" THEN 1

ELSE 0 // must be "ack" due to DOMAIN

$]$

# Choose Operator

CHOOSE $x$: $F$

expression that evaluates to some (possibly unspecific) value $x$ that satisfies $F$ // $x$ is a free variable in $F$

CHOOSE $x \in S$: $F \triangleq$ CHOOSE $x$: $x \in S \wedge F$

Undefined if no such $x$ exists

Example: $max(S) \triangleq$ CHOOSE $x \in S$: $\forall y \in S : x \geq y$

the maximum element of $S$ // undefined if $S$ is empty

# Choose Operator, cont'd

CHOOSE $x$: $F$ always evaluates to the same value. That is,

$$F \equiv G \Rightarrow (\text{CHOOSE } x\text{: } F) = (\text{CHOOSE } x\text{: } G)$$

Also

$$(v = \text{CHOOSE } x\text{: } F) \land (w = \text{CHOOSE } x\text{: } F) \Rightarrow v = w$$

However, the value of $v$ is unspecified

Q1: what behaviors are allowed by

$$(x = \text{CHOOSE } n\text{: } n \in Nat) \land \Box[x' = \text{CHOOSE } n\text{: } n \in Nat]_x \text{ ?}$$

# Choose Operator, cont'd

CHOOSE $x$: $F$ always evaluates to the same value. That is,

$$F \equiv G \Rightarrow (\text{CHOOSE } x\text{: } F) = (\text{CHOOSE } x\text{: } G)$$

Also

$$(v = \text{CHOOSE } x\text{: } F) \land (w = \text{CHOOSE } x\text{: } F) \Rightarrow v = w$$

However, the value of $v$ is unspecified

Q1: what behaviors are allowed by

$$(x = \text{CHOOSE } n\text{: } n \in Nat) \land \Box[x' = \text{CHOOSE } n\text{: } n \in Nat]_x \text{ ?}$$

Answer: $x$ is always the same (but unspecified) natural number

# Choose Operator, cont'd

CHOOSE $x\colon\ F$ always evaluates to the same value. That is,

$$F \equiv G \Rightarrow (\text{CHOOSE } x\colon\ F) = (\text{CHOOSE } x\colon G)$$

Also

$$(v = \text{CHOOSE } x\colon\ F) \wedge (w = \text{CHOOSE } x\colon\ F) \Rightarrow v = w$$

However, the value of $v$ is unspecified

Q1: what behaviors are allowed by

$$(x = \text{CHOOSE } n\colon n \in Nat) \wedge \Box[x' = \text{CHOOSE } n\colon n \in Nat]_x \text{ ?}$$

Answer: $x$ is always the same (but unspecified) natural number

Q2: what behaviors are allowed by

$$(x \in Nat) \wedge \Box[x' \in Nat]_x \text{ ?}$$

# Choose Operator, cont'd

CHOOSE $x$: $F$ always evaluates to the same value. That is,

$$F \equiv G \Rightarrow (\text{CHOOSE } x\text{: } F) = (\text{CHOOSE } x\text{: } G)$$

Also

$$(v = \text{CHOOSE } x\text{: } F) \wedge (w = \text{CHOOSE } x\text{: } F) \Rightarrow v = w$$

However, the value of $v$ is unspecified

Q1: what behaviors are allowed by

$$(x = \text{CHOOSE } n\text{: } n \in Nat) \wedge \Box[x' = \text{CHOOSE } n\text{: } n \in Nat]_x \text{ ?}$$

Answer: $x$ is always the same (but unspecified) natural number

Q2: what behaviors are allowed by

$$(x \in Nat) \wedge \Box[x' \in Nat]_x \text{ ?}$$

Answer: $x$ can be a different natural number in every state

# Recursive functions

- $fact \triangleq [n \in Nat \mapsto \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * fact[n - 1]]$

  is illegal because $fact$ is not defined in the expression on the right

Instead:

- $fact \triangleq CHOOSE\ f\colon [n \in Nat \mapsto \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * f[n - 1]]$

Shorthand:

- $fact[n \in Nat] \triangleq \text{IF } x = 0 \text{ THEN } 1 \text{ ELSE } n * fact[n - 1]$

# Aside: Scoping Definitions in TLA+

LET

$d_1 \triangleq e_1$

$d_2 \triangleq e_2$

…

IN

…

- Split complicated formulas into smaller chunks
- Leverage common subexpressions

# Representing an *n*-bit value

- We can represent an *n*-bit value by a function

$$b \triangleq [x \in 0..(n-1) \mapsto 0..1]$$

- For example, if $b$ represents 0101 (i.e., 5) then
  - $b[0] = 1$
  - $b[1] = 0$
  - $b[2] = 1$
  - $b[3] = 0$

and $b$ corresponds to the number

$$b[0] * 2^0 + b[1] * 2^1 + b[2] * 2^2 + b[3] * 2^3$$

# Finally: function of *n*-bit value *b* to number

$BitArrayVal(b) \triangleq$

LET

$n \triangleq$ CHOOSE m $\in Nat$: $\mathrm{DOMAIN}\ b = 0..(m-1)$

$f[x \in 0..(n-1)] \triangleq$

$\mathrm{IF}\ x = 0\ \mathrm{THEN}\ b[0]\ \mathrm{ELSE}\ b[x] * 2^x + f[x-1]$

IN

$f[n-1]$

# BinaryHourClock: (broken) attempt 1

$$
\begin{array}{l}
\text{MODULE } \mathit{BinaryHourClock} \\
\text{EXTENDS } \mathit{Naturals} \\
\text{VARIABLE } \mathit{bits} \\
\\
\mathit{BitArrayVal}(b) \triangleq \\
\quad \text{LET} \\
\qquad n \triangleq \text{CHOOSE } m \in \mathit{Nat} : \text{DOMAIN } b = 0 \,..\, (m-1) \\
\qquad f[x \in 0 \,..\, (n-1)] \triangleq \text{IF } x = 0 \text{ THEN } b[0] \text{ ELSE } b[x] * 2^x + f[x-1] \\
\quad \text{IN} \quad f[n-1] \\
\\
B \triangleq \text{INSTANCE } \mathit{HourClock} \text{ WITH } \mathit{hr} \leftarrow \mathit{BitArrayVal}(\mathit{bits}) \\
\\
\mathit{Spec} \triangleq B!\mathit{HC}
\end{array}
$$

Substitute *BitArrayVal*(*bits*) for *hr*

# What's the (subtle) issue?

- $BitArrayVal(b)$ is undefined unless b is a function $b$ with domain $0..n-1$ for some $n$
- $BitArrayVal(\text{“Fred”})$ is undefined
- Perhaps $BitArrayVal(\text{“Fred”})$ = 7. If so, “Fred” would be an allowed initial value of $bits$. Probably not what we intended to specify

Fix:

$$HourVal(b) \triangleq \text{IF } b \in [(0\,..\,3) \rightarrow (0\,..\,1)] \text{ THEN } BitArrayVal(b) \text{ ELSE } 13$$

$$B \triangleq \text{INSTANCE } HourClock \text{ WITH } hr \leftarrow HourVal(bits)$$

$$Spec \triangleq B!HC$$

Any value but 1..12

Because *HC* is never satisfied by a state in which *hr* = 13, *bits* has to be in [0..3 → 0..1]

# A little more elegant solution

$$\text{MODULE } \mathit{BinaryHourClock}$$

$\text{EXTENDS } \mathit{Naturals}$

$\text{VARIABLE } \mathit{bits}$

$$\begin{aligned}
&\mathit{BitArrayVal}(b) \triangleq \\
&\quad \text{LET} \\
&\qquad n \triangleq \text{CHOOSE } m \in \mathit{Nat} : \text{DOMAIN } b = 0 \,..\, (m-1) \\
&\qquad f[x \in 0 \,..\, (n-1)] \triangleq \text{IF } x = 0 \text{ THEN } b[0] \text{ ELSE } b[x] * 2^x + f[x-1] \\
&\quad \text{IN } f[n-1]
\end{aligned}$$

$$\boxed{\mathit{ErrorVal} \triangleq \text{CHOOSE } v : v \notin 1 \,..\, 12}$$

$$\mathit{HourVal}(b) \triangleq \text{IF } b \in [(0 \,..\, 3) \rightarrow (0 \,..\, 1)] \text{ THEN } \mathit{BitArrayVal}(b) \text{ ELSE } \boxed{\mathit{ErrorVal}}$$

$$B \triangleq \text{INSTANCE } \mathit{HourClock} \text{ WITH } hr \leftarrow \mathit{HourVal}(\mathit{bits})$$

$$\mathit{Spec} \triangleq B!HC$$

# A better way of doing it (instead of substitution)

MODULE *BinaryHourClock*

EXTENDS *Naturals*
VARIABLE *bits*

$H(hr) \triangleq$ INSTANCE *HourClock*

$BitArrayVal(b) \triangleq$
  LET
    $n \triangleq$ CHOOSE $m \in Nat$ : DOMAIN $b = 0 \,..\, (m-1)$
    $f[x \in 0 \,..\, (n-1)] \triangleq$ IF $x = 0$ THEN $b[0]$ ELSE $b[x] * 2^x + f[x-1]$
  IN $f[n-1]$

$ErrorVal \triangleq$ CHOOSE $v : v \notin 1 \,..\, 12$

$HourVal(b) \triangleq$ IF $b \in [(0 \,..\, 3) \rightarrow (0 \,..\, 1)]$ THEN $BitArrayVal(b)$ ELSE $ErrorVal$

$B \triangleq$ INSTANCE *HourClock* WITH $hr \leftarrow HourVal(bits)$

$IR(h) \triangleq \Box(h = HourVal(bits))$

$Spec \triangleq \exists\, hr : IR(hr) \wedge H(hr)!HC$

*bits* and hr *keep* the same time
(IR stands for Interface Refinement)

Hide *hr*

# Discussion

- Here we *composed* two specifications:
  - An HourClock with values from the set 1..12
  - A BinaryHourClock with values from the set [(0..3) → (0..1)]
- *Composition is a conjunction of specifications*
  - A behavior of the composition is a behavior of each of the components
- $IR(hr)$ asserts that $bits$ is always the 4-bit value representing $hr$
- $IR(hr) \wedge H(hr)!\, HC$ asserts that $hr$ and $bits$ keep the same time

More precisely:

- $IR(hr)$ defines $hr$ as a function of $bits$, but does not constrain $bits$
- $IR(hr) \wedge H(hr)!\, HC$ constrains behaviors involving $bits$ by requiring that they have to map to behaviors involving $hr$

# Interface Refinement

- BinaryHourClock is an *implementation* of HourClock
- One has to exhibit a mapping from the ”low-level” implementation to the “high-level” specification
  - Map the low-level state to the high-level state: $hr = HourVal(bits)$
  - Map each low-level step to a high-level step or to a high-level stuttering step
    - In the case of the BinaryHourClock, the low-level and high-level steps are the same
    - If so, that is a special case of interface refinement called “*Data Refinement*”
  - (leaving out liveness for now)
- Each behavior of the implementation is also a behavior of the specification

# We already saw an example of data refinement

MODULE *AsynchInterface*

EXTENDS *Naturals*
CONSTANT *Data*
VARIABLES *val*, *rdy*, *ack*

$$
\begin{aligned}
\mathit{TypeInvariant} \triangleq\ & \land\ \mathit{val} \in \mathit{Data} \\
& \land\ \mathit{rdy} \in \{0, 1\} \\
& \land\ \mathit{ack} \in \{0, 1\}
\end{aligned}
$$

---

$$
\begin{aligned}
\mathit{Init} \triangleq\ & \land\ \mathit{val} \in \mathit{Data} \\
& \land\ \mathit{rdy} \in \{0, 1\} \\
& \land\ \mathit{ack} = \mathit{rdy} \\
\mathit{Send} \triangleq\ & \land\ \mathit{rdy} = \mathit{ack} \\
& \land\ \mathit{val}' \in \mathit{Data} \\
& \land\ \mathit{rdy}' = 1 - \mathit{rdy} \\
& \land\ \text{UNCHANGED } \mathit{ack} \\
\mathit{Rcv} \triangleq\ & \land\ \mathit{rdy} \neq \mathit{ack} \\
& \land\ \mathit{ack}' = 1 - \mathit{ack} \\
& \land\ \text{UNCHANGED } \langle \mathit{val}, \mathit{rdy} \rangle \\
\mathit{Next} \triangleq\ & \mathit{Send} \lor \mathit{Rcv} \\
\mathit{Spec} \triangleq\ & \mathit{Init} \land \Box[\mathit{Next}]_{\langle \mathit{val}, \mathit{rdy}, \mathit{ack} \rangle}
\end{aligned}
$$

---

THEOREM $\mathit{Spec} \Rightarrow \Box \mathit{TypeInvariant}$

---

MODULE *Channel*

EXTENDS *Naturals*
CONSTANT *Data*
VARIABLE *chan*

$$
\mathit{TypeInvariant} \triangleq \mathit{chan} \in [\mathit{val} : \mathit{Data},\ \mathit{rdy} : \{0, 1\},\ \mathit{ack} : \{0, 1\}]
$$

---

$$
\begin{aligned}
\mathit{Init} \triangleq\ & \land\ \mathit{TypeInvariant} \\
& \land\ \mathit{chan.ack} = \mathit{chan.rdy} \\
\mathit{Send}(d) \triangleq\ & \land\ \mathit{chan.rdy} = \mathit{chan.ack} \\
& \land\ \mathit{chan}' = [\mathit{chan} \text{ EXCEPT } !.\mathit{val} = d,\ !.\mathit{rdy} = 1 - @] \\
\mathit{Rcv} \triangleq\ & \land\ \mathit{chan.rdy} \neq \mathit{chan.ack} \\
& \land\ \mathit{chan}' = [\mathit{chan} \text{ EXCEPT } !.\mathit{ack} = 1 - @] \\
\mathit{Next} \triangleq\ & (\exists\, d \in \mathit{Data} : \mathit{Send}(d)) \lor \mathit{Rcv} \\
\mathit{Spec} \triangleq\ & \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{chan}}
\end{aligned}
$$

---

THEOREM $\mathit{Spec} \Rightarrow \Box \mathit{TypeInvariant}$

# Interface Refinement with multiple steps

- Suppose we wanted a channel that sends values from 1..12
- And implement it over a channel that sends individual bits

High-level channel:

INSTANCE *Channel* WITH *Data* ← 1..12

Low-level channel:

INSTANCE *Channel* WITH *Data* ← 0..1

Represent each high-level value by sequence of four low-level bits

$$
\begin{array}{l}
\textsc{module } \mathit{Channel} \\
\hline
\textsc{extends } \mathit{Naturals} \\
\textsc{constant } \mathit{Data} \\
\textsc{variable } \mathit{chan} \\
\mathit{TypeInvariant} \triangleq \mathit{chan} \in [\mathit{val} : \mathit{Data},\ \mathit{rdy} : \{0,1\},\ \mathit{ack} : \{0,1\}] \\
\hline
\mathit{Init} \triangleq \land\ \mathit{TypeInvariant} \\
\qquad\qquad \land\ \mathit{chan.ack} = \mathit{chan.rdy} \\
\mathit{Send}(d) \triangleq \land\ \mathit{chan.rdy} = \mathit{chan.ack} \\
\qquad\qquad \land\ \mathit{chan}' = [\mathit{chan} \textsc{ except } !.\mathit{val} = d,\ !.\mathit{rdy} = 1 - @] \\
\mathit{Rcv} \triangleq \land\ \mathit{chan.rdy} \neq \mathit{chan.ack} \\
\qquad\qquad \land\ \mathit{chan}' = [\mathit{chan} \textsc{ except } !.\mathit{ack} = 1 - @] \\
\mathit{Next} \triangleq (\exists\, d \in \mathit{Data} : \mathit{Send}(d)) \lor \mathit{Rcv} \\
\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{chan}} \\
\hline
\textsc{theorem } \mathit{Spec} \Rightarrow \Box\, \mathit{TypeInvariant} \\
\hline
\end{array}
$$

# Channel Refinement

$$H \triangleq \text{INSTANCE } Channel \text{ WITH } chan \leftarrow h,\ Data \leftarrow 1 \,..\, 12$$
$$L \triangleq \text{INSTANCE } Channel \text{ WITH } chan \leftarrow l,\ Data \leftarrow \{0, 1\}$$

Sending 5 (= 0101):

$$s_0 \xrightarrow{L!\,Send(0)} s_1 \xrightarrow{L!\,Rcv} s_2 \xrightarrow{L!\,Send(1)} s_3 \xrightarrow{L!\,Rcv} s_4 \xrightarrow{L!\,Send(0)}$$
$$s_5 \xrightarrow{L!\,Rcv} s_6 \xrightarrow{L!\,Send(1)} s_7 \xrightarrow{L!\,Rcv} s_8 \longrightarrow \cdots$$

# Channel Refinement

$$H \triangleq \text{INSTANCE } \mathit{Channel} \text{ WITH } \mathit{chan} \leftarrow h,\ \mathit{Data} \leftarrow 1 \,..\, 12$$

$$L \triangleq \text{INSTANCE } \mathit{Channel} \text{ WITH } \mathit{chan} \leftarrow l,\ \mathit{Data} \leftarrow \{0, 1\}$$

Sending 5 (= 0101):

$$s_0 \xrightarrow{L!\mathit{Send}(0)} s_1 \xrightarrow{L!\mathit{Rcv}} s_2 \xrightarrow{L!\mathit{Send}(1)} s_3 \xrightarrow{L!\mathit{Rcv}} s_4 \xrightarrow{L!\mathit{Send}(0)}$$

$$s_5 \xrightarrow{L!\mathit{Rcv}} s_6 \xrightarrow{L!\mathit{Send}(1)} s_7 \xrightarrow{L!\mathit{Rcv}} s_8 \longrightarrow \cdots$$

Corresponds to $H!\mathit{Send}(5)$

Corresponds to $H!\mathit{Rcv}$

# Interface Refinement

Recall definition of $IR$ for BinaryHourClock a few slides ago:

- $IR$ will specify $h$ as a function of $l$, but does not constrain $l$

Then, if $HSpec$ is a high-level spec of the system, we can write the low-level spec as

$$\exists\, h \;:\; IR \wedge HSpec$$

MODULE *ChannelRefinement*

EXTENDS *Naturals*, *Sequences*

VARIABLES $h,\ l$

$ErrorVal \triangleq \text{CHOOSE } v : v \notin [val : 1 \,..\, 12,\ rdy : \{0, 1\},\ ack : \{0, 1\}]$

$BitSeqToNat[s \in Seq(\{0, 1\})] \triangleq$ $BitSeqToNat[\langle b_0, b_1, b_2, b_3 \rangle] = b_0 + 2 * (b_1 + 2 * (b_2 + 2 * b_3))$

$\quad \text{IF } s = \langle \rangle \text{ THEN } 0 \text{ ELSE } Head(s) + 2 * BitSeqToNat[Tail(s)]$

$H \triangleq \text{INSTANCE } Channel \text{ WITH } chan \leftarrow h,\ Data \leftarrow 1 \,..\, 12$

$L \triangleq \text{INSTANCE } Channel \text{ WITH } chan \leftarrow l,\ Data \leftarrow \{0, 1\}$

$H$ is a channel for sending numbers in $1 \,..\, 12$; $L$ is a channel for sending bits.

MODULE *Inner*

VARIABLE *bitsSent* — The sequence of the bits sent so far for the current number.

$$
\begin{aligned}
Init \triangleq\ & \land\ bitsSent = \langle\,\rangle \\
& \land\ \textsc{if}\ L!Init\ \textsc{then}\ H!Init \\
& \qquad\qquad\quad\ \textsc{else}\ h = ErrorVal
\end{aligned}
$$

Defines the initial value of $h$ as a function of $l$.

$$
\begin{aligned}
SendBit \triangleq\ & \exists\, b \in \{0,1\} : \\
& \quad \land\ L!Send(b) \\
& \quad \land\ \textsc{if}\ Len(bitsSent) < 3 \\
& \qquad\quad \textsc{then}\ \land\ bitsSent' = \langle b \rangle \circ bitsSent \\
& \qquad\qquad\qquad \land\ \textsc{unchanged}\ h \\
& \qquad\quad \textsc{else}\ \land\ bitsSent' = \langle\,\rangle \\
& \qquad\qquad\qquad \land\ H!Send(BitSeqToNat[\langle b \rangle \circ bitsSent])
\end{aligned}
$$

Sending one of the first three bits on $l$ prepends it to the front of *bitsSent* and leaves $h$ unchanged; sending the fourth bit resets *bitsSent* and sends the complete number on $h$.

$$
\begin{aligned}
RcvBit \triangleq\ & \land\ L!Rcv \\
& \land\ \textsc{if}\ bitsSent = \langle\,\rangle\ \textsc{then}\ H!Rcv \\
& \qquad\qquad\qquad\qquad\quad \textsc{else}\ \textsc{unchanged}\ h \\
& \land\ \textsc{unchanged}\ bitsSent
\end{aligned}
$$

A *Rcv* action on $l$ causes a *Rcv* action on $h$ iff it follows the sending of the fourth bit.

$$
\begin{aligned}
Error \triangleq\ & \land\ l' \neq l \\
& \land\ \neg((\exists\, b \in \{0,1\} : L!Send(b)) \lor L!Rcv) \\
& \land\ h' = ErrorVal
\end{aligned}
$$

An illegal action on $l$ sets $h$ to *ErrorVal*.

$$Next \triangleq SendBit \lor RcvBit \lor Error$$

$$InnerIR \triangleq Init \land \Box[Next]_{\langle l,\, h,\, bitsSent \rangle}$$

Note: standard TLA+ spec

MODULE *ChannelRefinement*

MODULE *Inner*

VARIABLE *bitsSent* The sequence of the bits sent so far for the current number.

$InnerIR \triangleq Init \land \Box[Next]_{\langle l, h, bitsSent \rangle}$ on previous page

$I(bitsSent) \triangleq$ INSTANCE *Inner*

$IR \triangleq \exists\, bitsSent : I(bitsSent)!InnerIR$

*h* is a function of *l*

MODULE *LowerSpec*

VARIABLE *lchan*

CONSTANT *Data*

$HS(hchan) \triangleq$ INSTANCE *Channel* WITH $chan \leftarrow hchan,\ Data \leftarrow 1 \,..\, 12$

$CR(h) \triangleq$ INSTANCE *ChannelRefinement* WITH $l \leftarrow lchan$

$LSpec \triangleq \exists\, h : CR(h)!IR \land HS(h)!HSpec$

constrains *lchan*

# Other examples of refinement

# Class Queue

```
class {:autocontracts} Queue {
  ghost var Contents: seq<int>;
  var a: array<int>;
  var hd: int, tl: int;

  predicate Valid() {              // class invariant
    a.Length > 0 && 0 <= tl <= hd <= a.Length && Contents == a[tl..hd]
  }

  constructor ()   ensures Contents == []
  {
    a, tl, hd, Contents := new int[10], 0, 0, [];
  }
}
```

# Class Queue: continued

```
method Enqueue(d: int)     ensures Contents == old(Contents) + [d]    {
    if hd == a.Length {
        var b := a;
        if tl == 0 { b := new int[2 * a.Length]; }                    // a is full
        forall (i | 0 <= i < hd – tl)  { b[i] := a[tl + i]; }         // shift
        a, tl, hd := b, 0, hd – tl;
    }
    a[hd], hd, Contents := d, hd + 1, Contents + [d];
  }

  method Dequeue() returns (d: int)
    requires Contents != []
    ensures d == old(Contents)[0] && Contents == old(Contents)[1..];
  {
    d, tl, Contents := a[tl], tl + 1, Contents[1..];
  }
```

# Chain Replication for Supporting High Throughput and Availability

Robbert van Renesse
rvr@cs.cornell.edu

Fred B. Schneider
fbs@cs.cornell.edu

*FAST Search & Transfer ASA*
*Tromsø, Norway*
*and*
*Department of Computer Science*
*Cornell University*
*Ithaca, New York 14853*

## Abstract

Chain replication is a new approach to coordinating clusters of fail-stop storage servers. The approach is intended for supporting large-scale storage services that exhibit high throughput and availability without sacrificing strong consistency guarantees. Besides outlining the chain replication protocols themselves, simulation experiments explore the performance characteristics of a prototype implementation. Throughput, availability, and several object-placement strategies (including schemes based on distributed hash table routing) are discussed.

## 1 Introduction

A *storage system* typically implements operations so that clients can store, retrieve, and/or change data. File systems and database systems are perhaps the best known examples. With a file system, operations (read and write) access a single file and are idempotent; with a database system, operations (transactions) may each access multiple objects and are serializable.

This paper is concerned with storage systems that sit somewhere between file systems and database systems. In particular, we are concerned with storage systems, henceforth called *storage services*, that

- store *objects* (of an unspecified nature),
- support *query* operations to return a value derived from a single object, and
- support *update* operations to atomically change the state of a single object according to some pre-programmed, possibly non-deterministic, computation involving the prior state of that object.

A file system write is thus a special case of our storage service update which, in turn, is a special case of a database transaction.

Increasingly, we see on-line vendors (like Amazon.com), search engines (like Google's and FAST's), and a host of other information-intensive services provide value by connecting large-scale storage systems to networks. A storage service is the appropriate compromise for such applications, when a database system would be too expensive and a file system lacks rich enough semantics.

One challenge when building a large-scale storage service is maintaining high availability and high throughput despite failures and concomitant changes to the storage service's configuration, as faulty components are detected and replaced.

Consistency guarantees also can be crucial. But even when they are not, the construction of an application that fronts a storage service is often simplified given *strong consistency guarantees*, which assert that (i) operations to query and update individual objects are executed in some sequential order and (ii) the effects of update operations are necessarily reflected in results returned by subsequent query operations.

Strong consistency guarantees are often thought to be in tension with achieving high throughput and high availability. So system designers, reluctant to sacrifice system throughput or availability, regularly decline to support strong consistency guarantees. The Google File System (GFS) illustrates this thinking [11]. In fact, strong consistency guarantees



**State is:**

$Hist_{objID}$ : **update request sequence**
$Pending_{objID}$ : **request set**

**Transitions are:**

T1: Client request $r$ arrives:
$Pending_{objID} := Pending_{objID} \cup \{r\}$

T2: Client request $r \in Pending_{objID}$ ignored:
$Pending_{objID} := Pending_{objID} - \{r\}$

T3: Client request $r \in Pending_{objID}$ processed:
$Pending_{objID} := Pending_{objID} - \{r\}$
**if** $r = \mathsf{query}(objId, opts)$ **then**
**reply** according options *opts* based on $Hist_{objID}$
**else if** $r = \mathsf{update}(objId, newVal, opts)$ **then**
$Hist_{objID} := Hist_{objID} \cdot r$
**reply** according options *opts* based on $Hist_{objID}$

Figure 1: Client's View of an Object.

### 3.1 Protocol Details

Clients do not directly read or write variables $Hist_{objID}$ and $Pending_{objID}$ of Figure 1, so we are free to implement them in any way that is convenient. When chain replication is used to implement the specification of Figure 1:

- $Hist_{objID}$ is defined to be $Hist^T_{objID}$, the value of $Hist_{objID}$ stored by tail $T$ of the chain, and
- $Pending_{objID}$ is defined to be the set of client requests received by any server in the chain and not yet processed by the tail.

The chain replication protocols for query processing and update processing are then shown to satisfy the specification of Figure 1 by demonstrating how each state transition made by any server in the chain is equivalent either to a no-op or to allowed transitions T1, T2, or T3.

**State is:**
$Hist_{objID}$ : **update request sequence**
$Pending_{objID}$ : **request set**

**Transitions are:**
T1: Client request $r$ arrives:
$Pending_{objID} := Pending_{objID} \cup \{r\}$

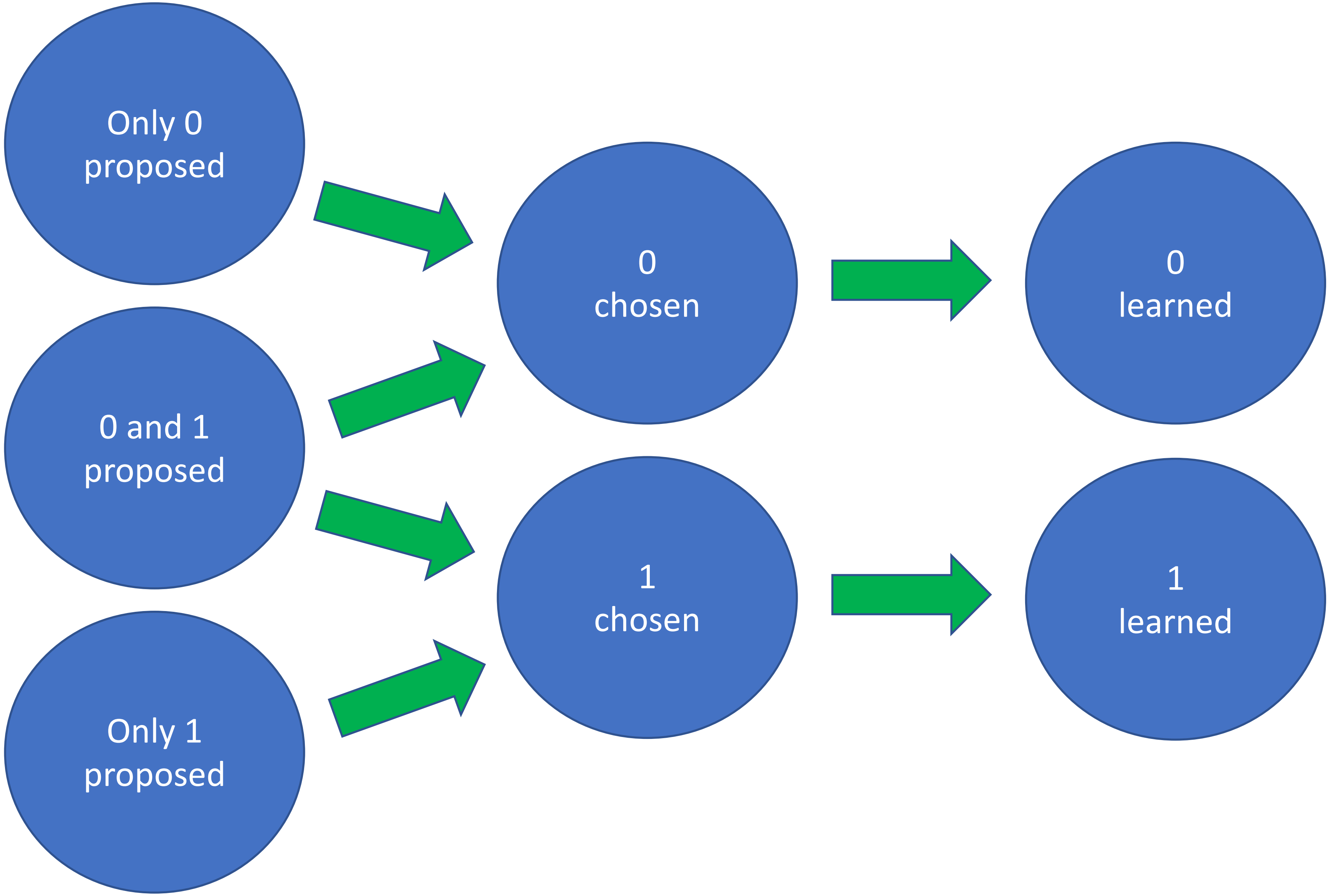
T2: Client request $r \in Pending_{objID}$ ignored:
$Pending_{objID} := Pending_{objID} - \{r\}$

T3: Client request $r \in Pending_{objID}$ processed:
$Pending_{objID} := Pending_{objID} - \{r\}$
**if** $r = \mathsf{query}(objId, opts)$ **then**
**reply** according options *opts* based on $Hist_{objID}$
**else if** $r = \mathsf{update}(objId, newVal, opts)$ **then**
$Hist_{objID} := Hist_{objID} \cdot r$
**reply** according options *opts* based on $Hist_{objID}$

Figure 1: Client's View of an Object.

# It’s not always possible to get a refinement ☹

# Binary Consensus, Specification



# Paxos

- Value is chosen if a quorum of proposers have all accepted the value on the same ballot
- This suggest an easy mapping of the Paxos state to the consensus state

# Problem 1: lack of history

- Unfortunately, Paxos acceptors only remember the latest value they accepted
- So while there may exists a majority that have all accepted the value at time t, that majority may no longer exist at time t+1
  - Even though it is guaranteed that no other value will ever be chosen

# Fix 1: add history variables

- We can add a “ghost variable” to each acceptor that remembers all (value, ballot) pairs it has ever accepted
  - “ghost” means that it does not actually have to be realized
- With this “history variable”, we can exhibit a state mapping

# Problem 2: outrunning the specification

- A refinement mapping maps each step of the low-level specification to either one step of the high-level specification or a stuttering step of the high-level specification
- In Paxos, when f=1 and n=3, the following scenario is possible:
  - Leader proposes a (value, ballot)
  - Some acceptor accepts (value, ballot)
  - In that one step:
    - The value is chosen
    - The acceptor learns that the value is chosen (decided)
- However, our high-level consensus spec requires two steps:
  - From undecided to chosen and from chosen to learned

# Fix 2: two possibilities

- Change the high-level spec to include a “choose + learn” step
  - i.e., speed up the high-level spec
  - complicates the high-level specification
  - changing the specification may not be allowed
- Add a ghost “prophecy variable” to the low-level specification
  - slow down the low-level spec
  - artificially insert a step between accepting and learning by changing the prophecy variable
  - does not change either the implementation or the high-level spec

# Completeness

- If S1 implements S2 then, possibly by adding history and prophecy variables, there exists a refinement mapping from S2 to S1 (under certain reasonable assumptions)

See Martin Abadi and Leslie Lamport, “The Existence of Refinement Mappings”