Specifying Systems (using TLA+)

Based on Leslie Lamport's book "Specifying Systems"

Definition: *State*

- Definition: A *state* is an assignment of values to (*all*) variables
- TLA+ notation: $[var_1 = value_1, var_2 = value_2, \cdots]$
  - Meaning: a state in which var_1 has value $value_1, \cdots$
  - Order is immaterial
- Example: $[hr = 3]$
  - Meaning: a state in which $hr = 3$
    - The values of other variables are not specified
  - There can be many infinitely many states in which $hr = 3$
    - e.g. $[hr = 3. temp = 62]$, $[hr = 3. temp = 68]$, …
  - *Models* perhaps the hour hand being 3 on some hour clock HC

Definition: *Behavior*

- Definition 1: A *behavior* is a function of time to state

Computer systems can be thought of as executing in steps, so

- Definition 2: 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*

- Definition: 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 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

Set of Initial States

- Example: HCini $\triangleq hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
  - Or, informally, HCini $\triangleq hr \in \{1, \cdots, 12\}$
  - HCini is simply a name given to the predicate
- A set of states can often be succinctly described by a predicate
  - Example: HCini $\triangleq hr \in \mathbb{N} \land 1 \leq hr \land hr \leq 12$
- Note again that these describe not 12 but an infinite set of states

Definition: *Next-State Relation*

- A *next-state relation* is a relation between pairs of successive states
  - $\{(state_1^{pre}, state_1^{post}), (state_2^{pre}, state_2^{post}), \cdots\}$
- Example:
  - HCnxt $\triangleq$ $\{\ ([hr = 11], [hr = 12]), ([hr = 12], [hr = 1]), \cdots\ \}$

Definition: *Action*

- A next-state relation can often be more succinctly described by a predicate
- Definition 1: an *action* is a predicate over a pair of states
- Example: HCnxt $\triangleq hr' = hr \mathbin{\%} 12 + 1$ (% is the "modulo" operator)
  - or, $\text{HCnxt}_2 \triangleq hr' = \text{IF } hr = 12 \text{ THEN } 1 \text{ ELSE } hr + 1$
  - But note that $\text{HCnxt}_2 \not\equiv \text{HCnxt}$
- hr' is the value of hr in the new state; hr is the value in the old state
- Definition 2: an *action* is a predicate containing both primed and unprimed variables
- An ordinary predicate and does not have to be of the form "*x*' = f(*x*)"
  - Example: HCnxt $\triangleq hr' - hr = 1 \bmod 12$

Steps versus *Actions* versus *Execution*

- A *step* is a pair of states
- An *action* $\mathcal{A}$ is a predicate over steps
- We call a step that satisfies $\mathcal{A}$ an $\mathcal{A}$ step
    - Example: a step that satisfies HCnxt is an *HCnxt step*
- We sometimes informally say that HCnxt is *executed*

Example specification: hour clock (in complete isolation)

Module HourClock

Variable hr

- HCini $\triangleq hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
- HCnxt $\triangleq hr' = hr \bmod 12 + 1$
- HC $\triangleq$ HCini $\land$ $\Box$HCnxt

Temporal logic formula $\Box$P means that predicate P *always* holds

(thus HCnxt is *invariant* in HC)

Note:

1. All three statements are definitions, but the last one happens to constitute the full specification of the hour clock)
2. There is no conventional naming in TLA+, so pick names that are descriptive

Definition: *Stuttering steps*

- Clocks are usually part of a larger system
- They have more state variables than just the hour hand of the clock
- State changes must allow for hour hand not to change
  - Example: $[hr = 3. temp = 62] \rightarrow [hr = 3. temp = 63]$
- This is called a *stuttering step* of the clock
  - i.e., $hr' = hr$

Final specification: hardware clock

Module HourClock

- Variable hr
- HCini $\triangleq hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
- HCnxt $\triangleq hr' = hr \bmod 12 + 1$
- HC $\triangleq$ HCini $\wedge \Box$(HCnxt $\vee$ $(hr' = hr)$)

The latter can be abbreviated using the following TLA+ notation

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

($[\text{HCnxt}]_{hr}$ is pronounced ”square HCnxt sub hr”)

Definition: *theorem*

- Definition: in TLA+, a *theorem* of a specification 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 $\Rightarrow \Box$ HCini
- Proof: by induction on #steps

A note on variables and types

- Variables in TLA+ are untyped
- However, if one can prove SPEC $\Rightarrow \Box\, v \in S$ for some variable v and constant set S, then one can call S the type of v in SPEC
- Example: the type of hr in HC is $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
- It is useful to specify the types in a specification
- Example: HCtypeInvariant $\triangleq hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
- Note, in this case HCtypeInvariant $\equiv$ HCini

A note on states and behaviors

- Recall
  - A state is an assignment of values to variables
  - A behavior is a sequence of states
- Thus
  - $[hr = 13]$ is still a state, and so is $[hr = "blue"]$
  - $[hr = 4] \rightarrow [hr = 3]$ is still a behavior
- However, they are not in specification HC

Recall

- A *state* is an assignment of values to all variables
- A *step* is a pair of states
- A *stuttering step* wrt some variable leaves the variable unchanged
- An *action* is a predicate over a pair of states
  - If x is a variable in the old state, then x' is the same variable in the new state
- A *behavior* is an infinite sequence of states (with an initial state)
- A *specification* characterizes the initial state and actions

Spec that generates all prime numbers

MODULE *prime*

EXTENDS *Naturals*

VARIABLE p

$$isPrime(q) \triangleq q > 1 \land \forall r \in 2 \,..\, (q-1) : q \% r \neq 0$$

$$TypeInvariant \triangleq isPrime(p)$$

$$Init \triangleq p = 2$$

$$Next \triangleq p' > p \land isPrime(p') \land \forall q \in (p+1) \,..\, (p'-1) : \neg isPrime(q)$$

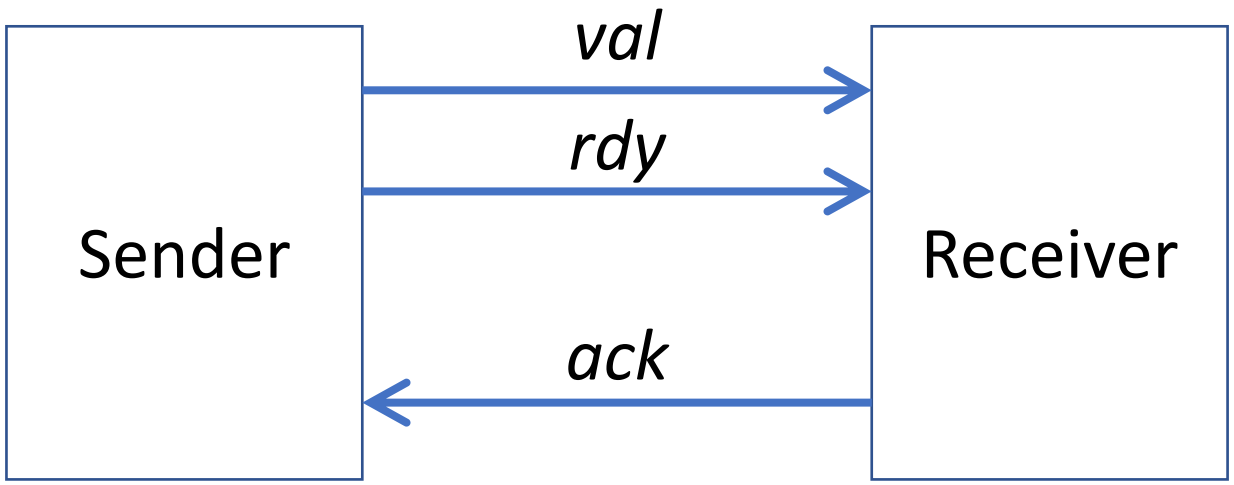
$$Spec \triangleq Init \land \Box[Next]_p$$

THEOREM $Spec \Rightarrow \Box\, TypeInvariant$

Spec that generates all prime numbers

```
------------------------------ MODULE prime ------------------------------
EXTENDS Naturals
VARIABLE p

isPrime(q) == q > 1 /\ \A r \in 2..(q-1): q%r /= 0

TypeInvariant == isPrime(p)

Init == p = 2
Next == p' > p /\ isPrime(p') /\ \A q \in (p+1)..(p'-1): ~isPrime(q)

Spec == Init /\ [] [Next]_p

THEOREM Spec => []TypeInvariant
```

Asynchronous FIFO Channel Specification



$$Send \triangleq \land\ rdy = ack$$
$$\land\ val' \in Data$$
$$\land\ rdy' = 1 - rdy$$
$$\land\ ack' = ack$$

$$Recv \triangleq \land\ rdy \neq ack$$
$$\land\ ack' = 1 - ack$$
$$\land\ val' = val$$
$$\land\ rdy' = rdy$$

Asynchronous FIFO Channel Specification

$$TypeInvariant \triangleq \land\ val \in Data \quad \land\ rdy \in \{0, 1\} \quad \land\ ack \in \{0, 1\}$$

$$Init \triangleq \land\ val \in Data \quad \land\ rdy \in \{0, 1\} \quad \land\ ack = rdy$$

$$Send \triangleq \land\ rdy = ack \quad \land\ val' \in Data \quad \land\ rdy' = 1 - rdy \quad \land\ ack' = ack$$

$$Recv \triangleq \land\ rdy \neq ack \quad \land\ ack' = 1 - ack \quad \land\ val' = val \quad \land\ rdy' = rdy$$

$$Next \triangleq Send \lor Recv$$

$$Spec \triangleq \text{Init} \land \Box[\text{Next}]_{\langle rdy, ack, val \rangle}$$

Asynchronous FIFO Channel Specification

introducing *operators with arguments*

$$Send \triangleq \land\ rdy = ack \quad \land\ val' \in Data \quad \land\ rdy' = 1 - rdy \quad \land\ ack' = ack$$

$\Rightarrow$

$$Send(d) \triangleq \land\ rdy = ack \quad \land\ val' = d \quad \land\ rdy' = 1 - rdy \quad \land\ ack' = ack$$

$$Next \triangleq \lor\ Send \quad \lor\ Recv$$

$\Rightarrow$

$$Next \triangleq \lor\ \exists d \in Data: Send(d) \quad \lor\ Recv$$

Asynchronous FIFO Channel Specification
introducing *records*

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

$Init \triangleq chan.val \in Data \land chan.rdy \in \{0, 1\} \land chan.ack = chan.rdy$

$Send(d) \triangleq chan.rdy = chan.ack \land chan' =$
$[val \mapsto d, rdy \mapsto 1 - chan.rdy, ack \mapsto chan.ack]$

$Recv \triangleq chan.rdy \neq chan.ack \land chan' =$
$[val \mapsto chan.val, rdy \mapsto chan.rdy, ack \mapsto 1 - chan.ack]$

$Next \triangleq \exists d \in Data: Send(d) \lor Recv$

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

Some more terms

- A *state function* is an ordinary expression with (unprimed) variables
  - i.e., it is a function of a state to a value
  - note that a variable is a state function
- A *state predicate* is a Boolean state function
- A *temporal formula* is an assertion about behaviors
- *A theorem* of a specification is a temporal formula that holds over every behavior of the specification
- 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.

Temporal Formula

Based on Chapter 8 of Specifying Systems

- A *temporal formula* F assigns a Boolean value to a behavior σ
- $\sigma \vDash F$ means that F holds over σ
- F is a theorem if $\sigma \vDash F$ holds over all behaviors σ
- If P is a state predicate, then $\sigma \vDash P$ means that P holds over the first state in σ
- If A is an action, then $\sigma \vDash A$ means that A holds over the first two states in σ
  - i.e., the first step in σ is an A step
- If A is an action, then $\sigma \vDash [A]_v$ means that the first step in σ is an A step or a stuttering step with respect to v

$\Box$Always

- $\sigma \vDash \Box F$ means that F holds over every suffix of σ
- More formally
  - Let σ^{+n} be σ with the first n states removed
  - Then $\sigma \vDash \Box F \triangleq \forall n \in \mathbb{N}\colon \sigma^{+n} \vDash F$

Boolean combinations of temporal formulas

- $\sigma \vDash (F \wedge G) \triangleq (\sigma \vDash F) \wedge (\sigma \vDash G)$
- $\sigma \vDash (F \vee G) \triangleq (\sigma \vDash F) \vee (\sigma \vDash G)$
- $\sigma \vDash \neg F \triangleq \neg (\sigma \vDash F)$
- $\sigma \vDash (F \Rightarrow G) \triangleq (\sigma \vDash F) \Rightarrow (\sigma \vDash G)$
- $\sigma \vDash (\exists r: F) \triangleq \exists r: \sigma \vDash F$
- $\sigma \vDash (\forall r \in S: F) \triangleq \forall r \in S: \sigma \vDash F$ // if S is a constant set

Example

What is the meaning of $\sigma \vDash \Box((x = 1) \Rightarrow \Box(y > 0))$?

$$\sigma \vDash \Box((x = 1) \Rightarrow \Box(y > 0))$$
$$\equiv \forall n \in \mathbb{N}\colon \sigma^{+n} \vDash ((x = 1) \Rightarrow \Box(y > 0))$$
$$\equiv \forall n \in \mathbb{N}\colon (\sigma^{+n} \vDash (x = 1)) \Rightarrow (\sigma^{+n} \vDash \Box(y > 0))$$
$$\equiv \forall n \in \mathbb{N}\colon (\sigma^{+n} \vDash (x = 1)) \Rightarrow (\forall m \in \mathbb{N}\colon (\sigma^{+n})^{+m} \vDash (y > 0))$$

If $x = 1$ in some state, then henceforth $y > 0$ in all subsequent states

Not: once x = 1, x will always be 1. That would be

$$\sigma \vDash \Box((x = 1) \Rightarrow \Box(x = 1))$$

Not every temporal formula is a TLA+ formula

- TLA+ formulas are temporal formulas that are *invariant under stuttering*
  - They hold even if you add or remove stuttering steps
- Examples
  - P if P is a state predicate
  - $\Box P$ if P is a state predicate
  - $\Box[A]_v$ if A is an action and v is a state variable (or even state function)
- But not
  - $x' = x + 1$ — not satisfied by $[x = 1] \longrightarrow [x = 1] \longrightarrow [x = 2]$
  - $[x' = x + 1\,]_x$ — satisfied by $[x = 1] \longrightarrow [x = 1] \longrightarrow [x = 3]$
    but not by $[x = 1] \longrightarrow [x = 3]$
- Yet $\Box[x' = x + 1\,]_x$ is a TLA+ formula!

Eventually F

$\Diamond F \triangleq \neg\Box\neg F$

$\sigma \vDash \Diamond F$

$\equiv \sigma \vDash \neg\Box\neg F$

$\equiv \neg(\sigma \vDash \Box\neg F)$

$\equiv \neg(\forall n \in \mathbb{N}:\ \sigma^{+n} \vDash \neg F)$

$\equiv \neg(\forall n \in \mathbb{N}:\ \neg(\sigma^{+n} \vDash F))$

$\equiv \exists n \in \mathbb{N}:\ (\sigma^{+n} \vDash F)$

Eventually an A step occurs that changes v…

$$\Diamond \langle A \rangle_v \triangleq \neg\Box[\neg A]_v$$

$$\sigma \vDash \Diamond \langle A \rangle_v$$

$$\equiv \sigma \vDash \neg\Box[\neg A]_v$$

$$\equiv \neg(\sigma \vDash \Box[\neg A]_v)$$

$$\equiv \neg(\forall n \in \mathbb{N}: \sigma^{+n} \vDash [\neg A]_v)$$

$$\equiv \neg(\forall n \in \mathbb{N}: \sigma^{+n} \vDash (\neg A \lor v' = v))$$

$$\equiv \exists n \in \mathbb{N}: \sigma^{+n} \vDash (A \land v' \neq v)$$

HourClock revisited

Module HourClock

Variable *hr*

hr is a parameter of the specification HourClock

- HCini ≜ $hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
- HCnxt ≜ $hr' = hr \bmod 12 + 1$
- HC ≜ HCini ∧ □[HCnxt]$_{hr}$

HourClock with *liveness*

clock that never stops

Module HourClock

Variable hr

- HCini $\triangleq$ $hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
- HCnxt $\triangleq$ $hr' = hr \bmod 12 + 1$
- HC $\triangleq$ HCini $\wedge$ $\square[\text{HCnxt}]_{hr}$
- LiveHC $\triangleq$ HC $\wedge$ $\square(\Diamond\langle \text{HCnxt} \rangle_{hr})$

Module Channel with Liveness

Constant $Data$ Variable $chan$

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

$Init \triangleq chan.val \in Data \land chan.rdy \in \{0, 1\} \land chan.ack = chan.rdy$

$Send(d) \triangleq chan.rdy = chan.ack \land chan' =$
$[val \longmapsto d, rdy \longmapsto 1 - chan.rdy, ack \longmapsto chan.ack]$

$Recv \triangleq chan.rdy \neq chan.ack \land chan' =$
$[val \longmapsto chan.val, rdy \longmapsto chan.rdy, ack \longmapsto 1 - chan.ack]$

$Next \triangleq \exists d \in Data: Send(d) \lor Recv$

$Spec \triangleq Init \land \Box[Next]_{chan}$ $LiveSpec \triangleq Spec \land \Box(\Diamond\langle Next\rangle_{chan})$???

Module Channel with Liveness

Constant $Data$ Variable $chan$

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

$Init \triangleq chan.val \in Data \land chan.rdy \in \{0, 1\} \land chan.ack = chan.rdy$

$Send(d) \triangleq chan.rdy = chan.ack \land chan' =$
$[val \mapsto d, rdy \mapsto 1 - chan.rdy, ack \mapsto chan.ack]$

$Recv \triangleq chan.rdy \neq chan.ack \land chan$
$[val \mapsto chan.val, rd$ $- chan.ack]$

Too Strong --- If nothing to send that should be ok

$Next \triangleq \exists d \in Data: Send(d) \lor Recv$

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

$LiveSpec \triangleq Spec \land \Box(\Diamond\langle Next\rangle_{chan})$???

Module Channel with Liveness

Constant $Data$ Variable $chan$

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

$Init \triangleq chan.val \in Data \land chan.rdy \in \{ 0, 1 \} \land chan.ack = chan.rdy$

$Send(d) \triangleq chan.rdy = chan.ack \land chan' =$
$[\, val \mapsto d, rdy \mapsto 1 - chan.rdy, ack \mapsto chan.ack \,]$

$Recv \triangleq chan.rdy \neq chan.ack \land chan' =$
$[\, val \mapsto chan.val, rdy \mapsto chan.rdy, ack \mapsto 1 - chan.ack \,]$

$Next \triangleq \exists d \in Data: Send(d) \lor Recv$

$Spec \triangleq Init \land \square[Next]_{chan}$

$LiveSpec \triangleq Spec \land \square(chan.rdy \neq chan.ack \Rightarrow \diamond\langle Recv \rangle_{chan})$

Weak Fairness as a liveness condition

- $\text{ENABLED}\ \langle A \rangle_v$ means action A is possible in some state
  - State predicate conjuncts all hold *and* some next state must exist
- $WF_v(A) \triangleq \Box(\Box\text{ENABLED}\ \langle A \rangle_v \Rightarrow \Diamond\langle A \rangle_v)$

- HourClock: $WF_{hr}(HCnxt)$
- Channel: $WF_{chan}(Recv)$

(surprising) Weak Fairness equivalence

- $WF_v(A) \triangleq \Box(\Box\text{ENABLED}\,\langle A\rangle_v \Rightarrow \Diamond\langle A\rangle_v)$

  $\equiv \Box\Diamond(\neg\text{ENABLED}\,\langle A\rangle_v) \vee \Box\Diamond\langle A\rangle_v$

  $\equiv \Diamond\Box(\text{ENABLED}\,\langle A\rangle_v) \Rightarrow \Box\Diamond\langle A\rangle_v$

- Always, if A is enabled forever, then an A step eventually occurs
- A is infinitely often disabled or infinitely many A steps occur
- If A is eventually enabled forever then infinitely many A steps occur

Strong Fairness

- $SF_v(A) \triangleq \Diamond\Box(\neg\text{ENABLED}\,\langle A\rangle_v) \lor \Box\Diamond\langle A\rangle_v$

  $\equiv \Box\Diamond(\text{ENABLED}\,\langle A\rangle_v) \Rightarrow \Box\Diamond\langle A\rangle_v$

- A is eventually disabled forever or infinitely many A steps occur
- If A is infinitely often enabled then infinitely many A steps occur

$SF_v(A)$: an A step must occur if A is continually enabled

$WF_v(A)$: an A step must occur if A is continuously enabled

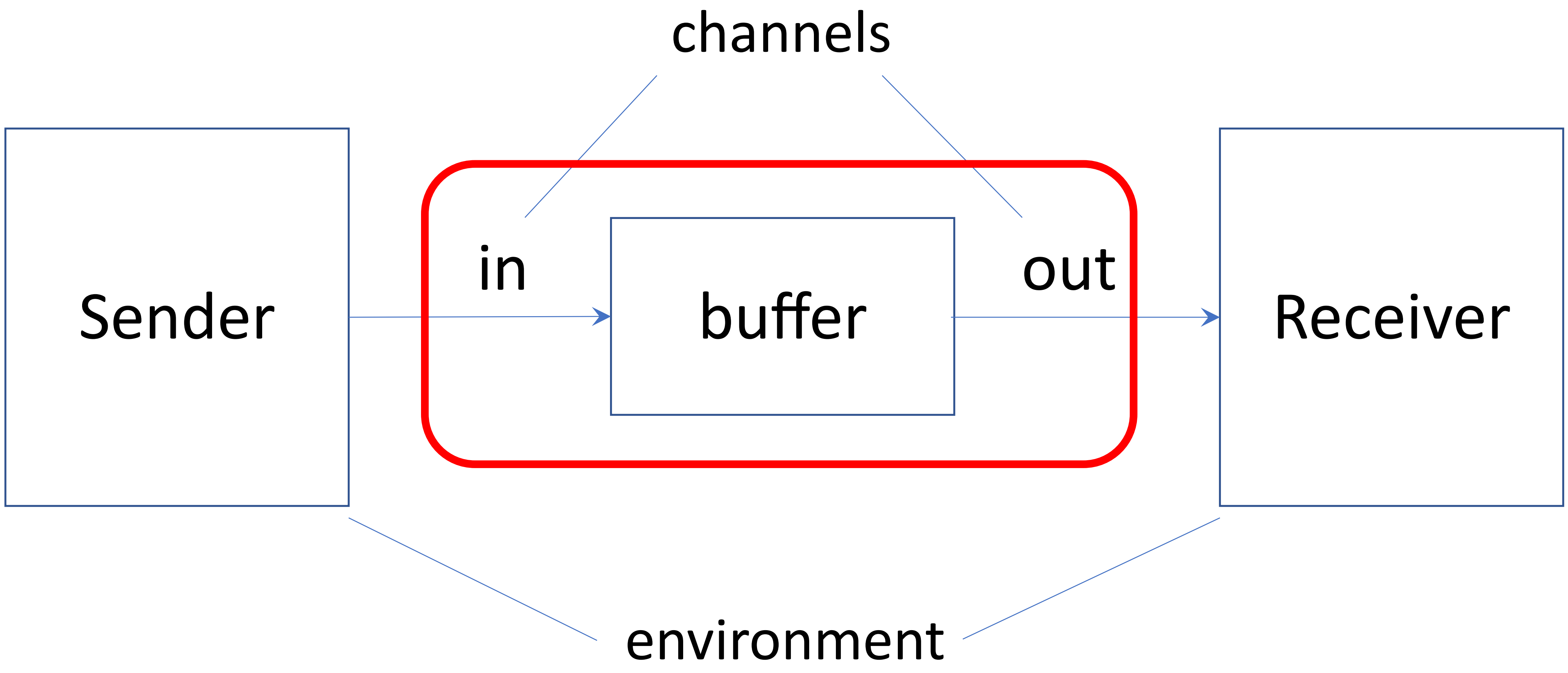
As always, better to make the weaker assumption if you can

How important is liveness?

- Liveness rules out behaviors that have only stuttering steps
  - Add non-triviality of a specification
- In practice, “eventual” is often not good enough
- Instead, need to specify performance requirements
  - Service Level Objectives (SLOs)
  - Usually done quite informally

A "FIFO" (async buffered FIFO channel)

Chapter 4 from Specifying Systems



Module Channel

Constant $\boldsymbol{Data}$ **Variable** $\boldsymbol{chan}$

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

Init $\triangleq chan.val \in Data \land chan.rdy \in \{0, 1\} \land chan.ack = chan.rdy$

Send(d) $\triangleq chan.rdy = chan.ack \land chan' =$
$[val \mapsto d, rdy \mapsto 1 - chan.rdy, ack \mapsto chan.ack]$

Recv $\triangleq chan.rdy \neq chan.ack \land chan' =$
$[val \mapsto chan.val, rdy \mapsto chan.rdy, ack \mapsto 1 - chan.ack]$

Next $\triangleq \exists d \in Data: Send(d) \lor Recv$

Spec $\triangleq Init \land \square[Next]_{chan}$

Instantiating a Channel

$InChan \triangleq$ INSTANCE $Channel$ WITH $Data \leftarrow Message, chan \leftarrow in$

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

$InChan!TypeInvariant \equiv in \in [val: Message, rdy: \{0,1\}, ack: \{0,1\}]$

Instantiation is Substitution!

MODULE *InnerFIFO*

EXTENDS *Naturals*, *Sequences*

CONSTANT *Message*

VARIABLES *in*, *out*, *q*

$InChan \triangleq$ INSTANCE *Channel* WITH $Data \leftarrow Message,\ chan \leftarrow in$

$OutChan \triangleq$ INSTANCE *Channel* WITH $Data \leftarrow Message,\ chan \leftarrow out$

$$
\begin{aligned}
Init \triangleq\ & \land\ InChan!Init \\
& \land\ OutChan!Init \\
& \land\ q = \langle\ \rangle
\end{aligned}
$$

$$
\begin{aligned}
TypeInvariant \triangleq\ & \land\ InChan!TypeInvariant \\
& \land\ OutChan!TypeInvariant \\
& \land\ q \in Seq(Message)
\end{aligned}
$$

$$SSend(msg) \triangleq \land InChan!Send(msg) \land \text{UNCHANGED } \langle out, q \rangle$$

Send *msg* on channel *in*.

$$BufRcv \triangleq \land InChan!Rcv \land q' = Append(q, in.val) \land \text{UNCHANGED } out$$

Receive message from channel *in* and append it to tail of *q*.

$$BufSend \triangleq \land q \neq \langle \rangle \land OutChan!Send(Head(q)) \land q' = Tail(q) \land \text{UNCHANGED } in$$

Enabled only if *q* is nonempty. Send *Head*(*q*) on channel *out* and remove it from *q*.

$$RRcv \triangleq \land OutChan!Rcv \land \text{UNCHANGED } \langle in, q \rangle$$

Receive message from channel *out*.

$$Next \triangleq \lor \exists\, msg \in Message : SSend(msg)$$
$$\qquad \lor BufRcv$$
$$\qquad \lor BufSend$$
$$\qquad \lor RRcv$$

$$Spec \triangleq Init \land \Box[Next]_{\langle in,\, out,\, q \rangle}$$

THEOREM $Spec \Rightarrow \Box TypeInvariant$

Parametrized Instantiation

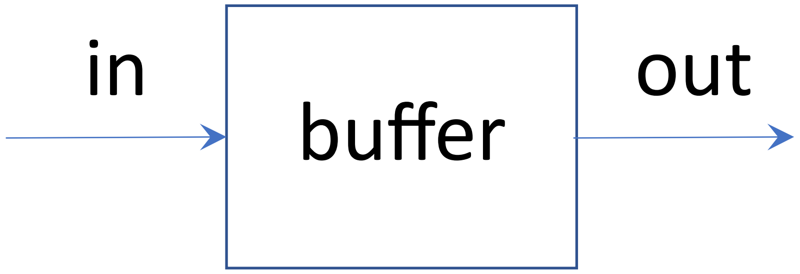
$InChan \triangleq$ INSTANCE $Channel$ WITH $Data \leftarrow Message, chan \leftarrow in$

$Chan(ch) \triangleq$ INSTANCE $Channel$ WITH $Data \leftarrow Message, chan \leftarrow ch$

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

$Chan(in)!TypeInvariant \equiv in \in [val: Message, rdy: \{0,1\}, ack: \{0,1\}]$

Internal (= Non-Interface) Variables



There is no *q* here

MODULE *InnerFIFO*

EXTENDS *Naturals*, *Sequences*
CONSTANT *Message*
VARIABLES *in*, *out*, *q*

But there is a *q* here

Not incorrect, but don't really want *q* to be a specification parameter

Hiding Internal Variables

MODULE *FIFO*

CONSTANT *Message*

VARIABLES *in*, *out*

$Inner(q) \triangleq$ INSTANCE *InnerFIFO*

$Spec \triangleq \exists\, q : Inner(q)!Spec$

Hiding Internal Variables

MODULE *FIFO*

CONSTANT *Message*

VARIABLES *in*, *out*

$Inner(q) \triangleq$ INSTANCE *InnerFIFO*

$Spec \triangleq \exists\kern-0.5em\exists\, q : Inner(q)!Spec$

Not the normal existential quantifier!!!

In temporal logic, this means that for every state in a behavior, there is a value for *q* that makes *Inner(q)!Spec* true

Pretty. Now for something cool!

- Suppose we wanted to implemented a bounded buffer
- That is, $\Box len(q) \leq N$ for some constant $N > 0$
- The only place where q is extended is in *BufRcv*

$$
\begin{aligned}
BufRcv \triangleq\ & \land\ InChan!Rcv \\
& \land\ q' = Append(q,\ in.val) \\
& \land\ \text{UNCHANGED}\ out
\end{aligned}
$$

Pretty. Now for something cool!

- Suppose we wanted to implemented a bounded buffer
- That is, $\Box len(q) \leq N$ for some constant $N > 0$
- The only place where q is extended is in *BufRcv*

$$
\begin{aligned}
BufRcv \triangleq\ & \land\ InChan!Rcv \\
& \land\ q' = Append(q, in.val) \\
& \land\ \text{UNCHANGED } out \\
& \land\ len(q) < N
\end{aligned}
$$

Even cooler (but tricky)

MODULE *BoundedFIFO*

EXTENDS *Naturals*, *Sequences*
VARIABLES *in*, *out*
CONSTANT *Message*, *N*
ASSUME $(N \in Nat) \land (N > 0)$

$Inner(q) \triangleq$ INSTANCE *InnerFIFO*

$$BNext(q) \triangleq \land\ Inner(q)!Next \\ \land\ Inner(q)!BufRcv \Rightarrow (Len(q) < N)$$

$$Spec \triangleq \exists\, q : Inner(q)!Init \land \Box[BNext(q)]_{\langle in, out, q \rangle}$$

If it is a *BufRcv* step, then $len(q) < N$

Even cooler (but tricky)

MODULE *BoundedFIFO*

EXTENDS *Naturals*, *Sequences*

VARIABLES *in*, *out*

CONSTANT *Message*, *N*

ASSUME $(N \in Nat) \land (N > 0)$

$Inner(q) \triangleq$ INSTANCE *InnerFIFO*

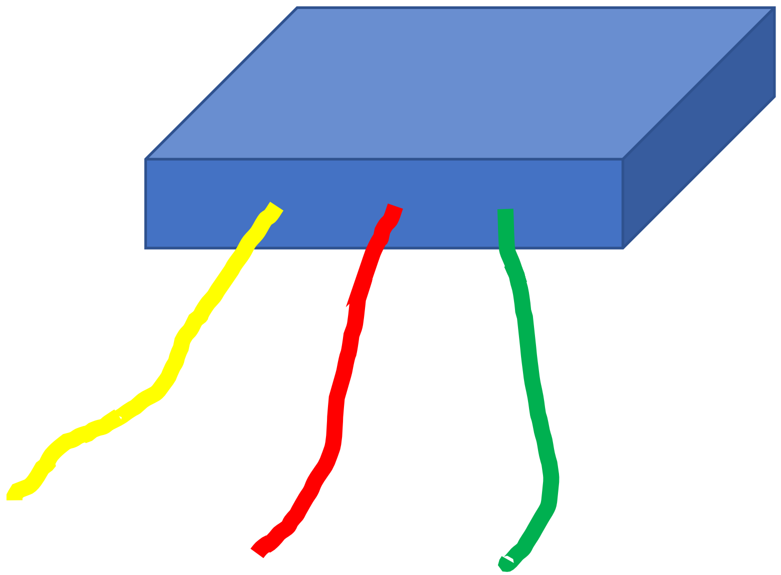
$$BNext(q) \triangleq \land\ Inner(q)!Next$$
$$\land\ Inner(q)!BufRcv \Rightarrow (Len(q) < N)$$

$$Spec \triangleq \exists\, q : Inner(q)!Init \land \Box[BNext(q)]_{\langle in, out, q \rangle}$$

Refinement

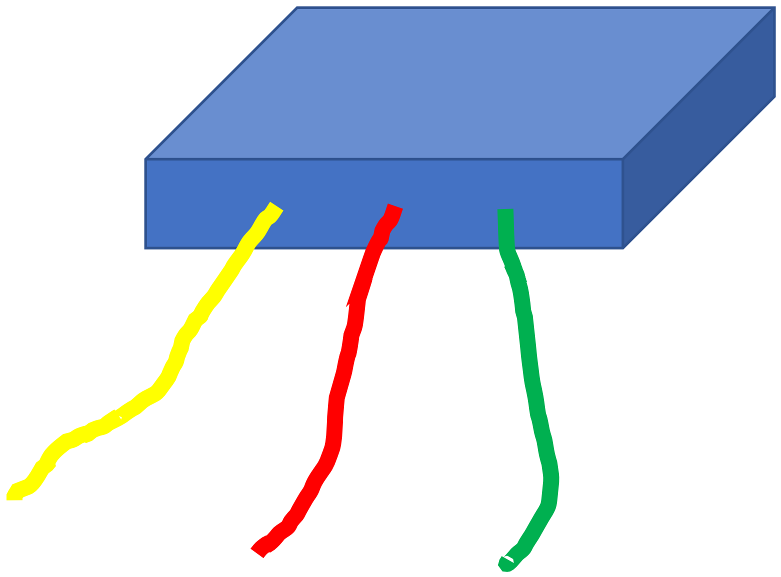
Based on material from Section 10.8, Specifying Systems by Leslie Lamport

You ask for:



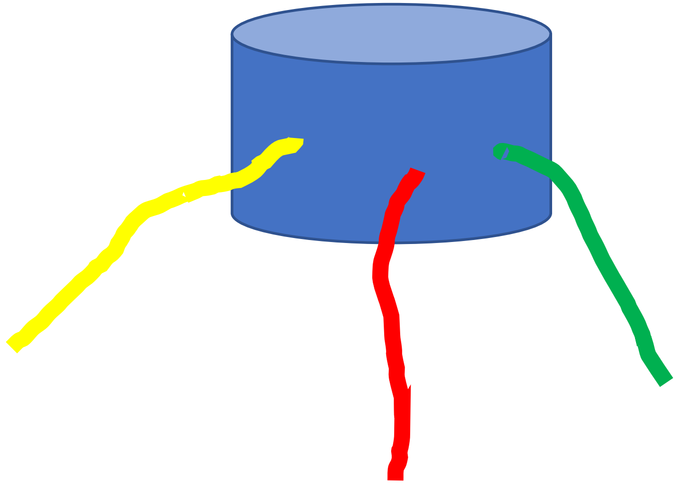
Specification

You ask for:



Specification

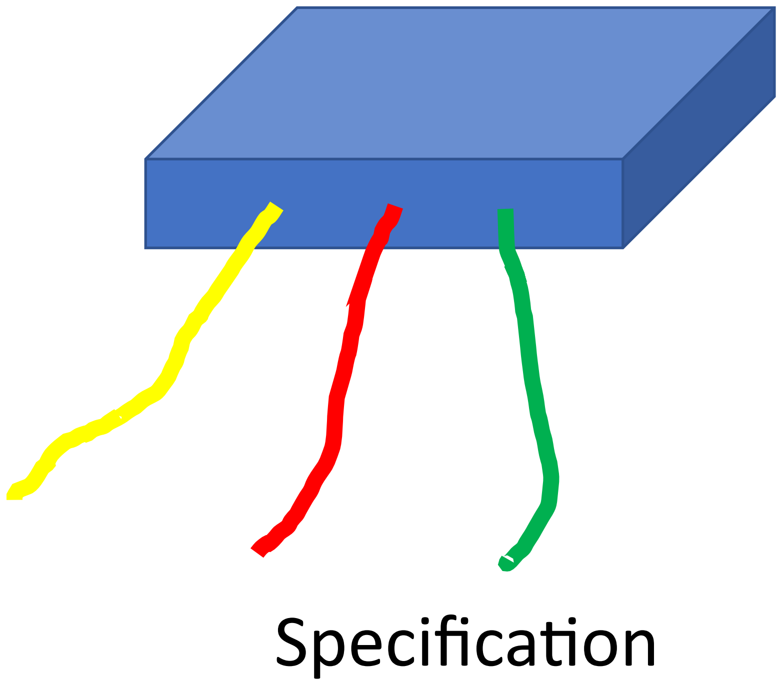
You get:

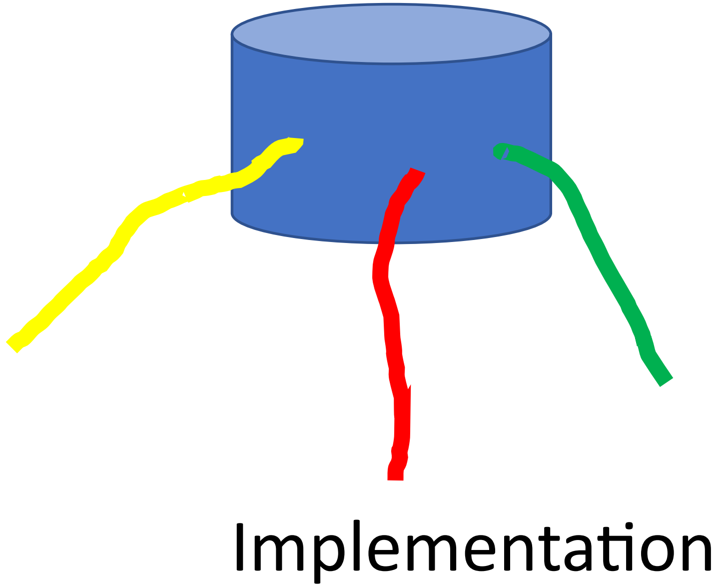


Implementation

You ask for:

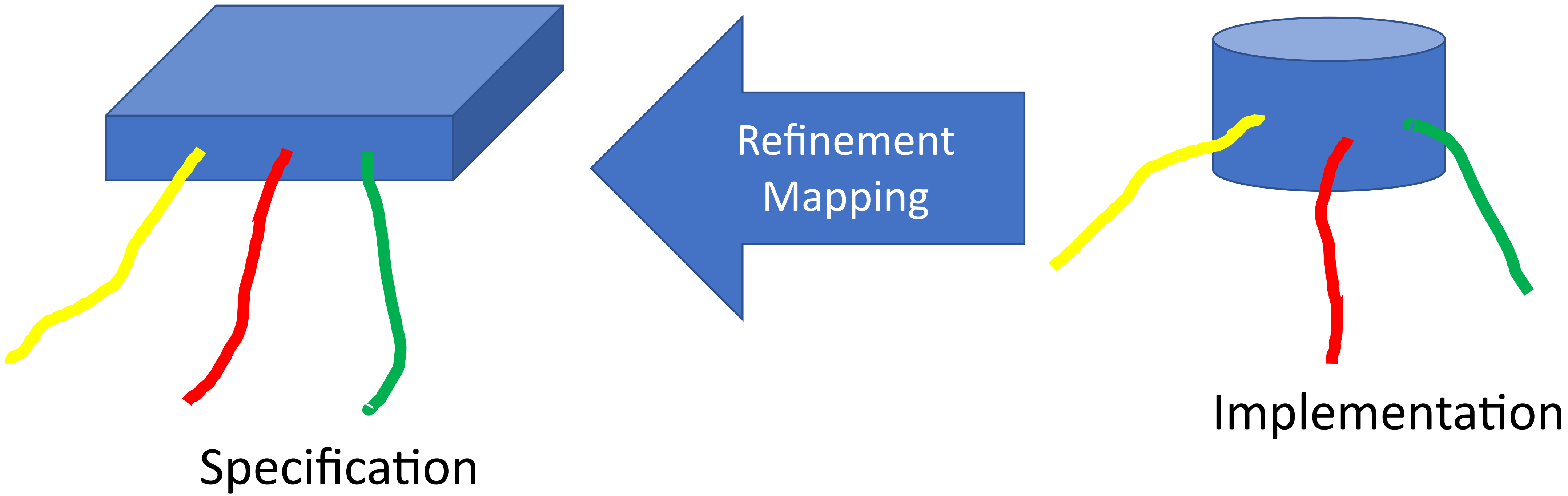
You get:





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

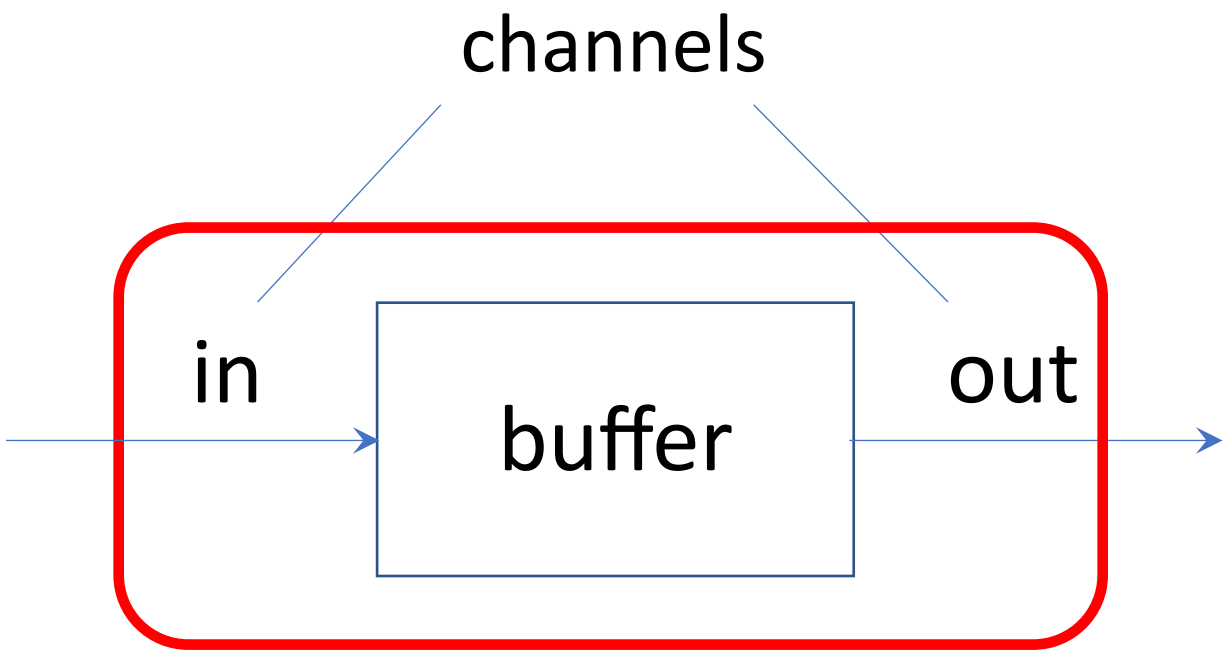
You ask for: You get:



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

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

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 (up to stuttering) 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

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

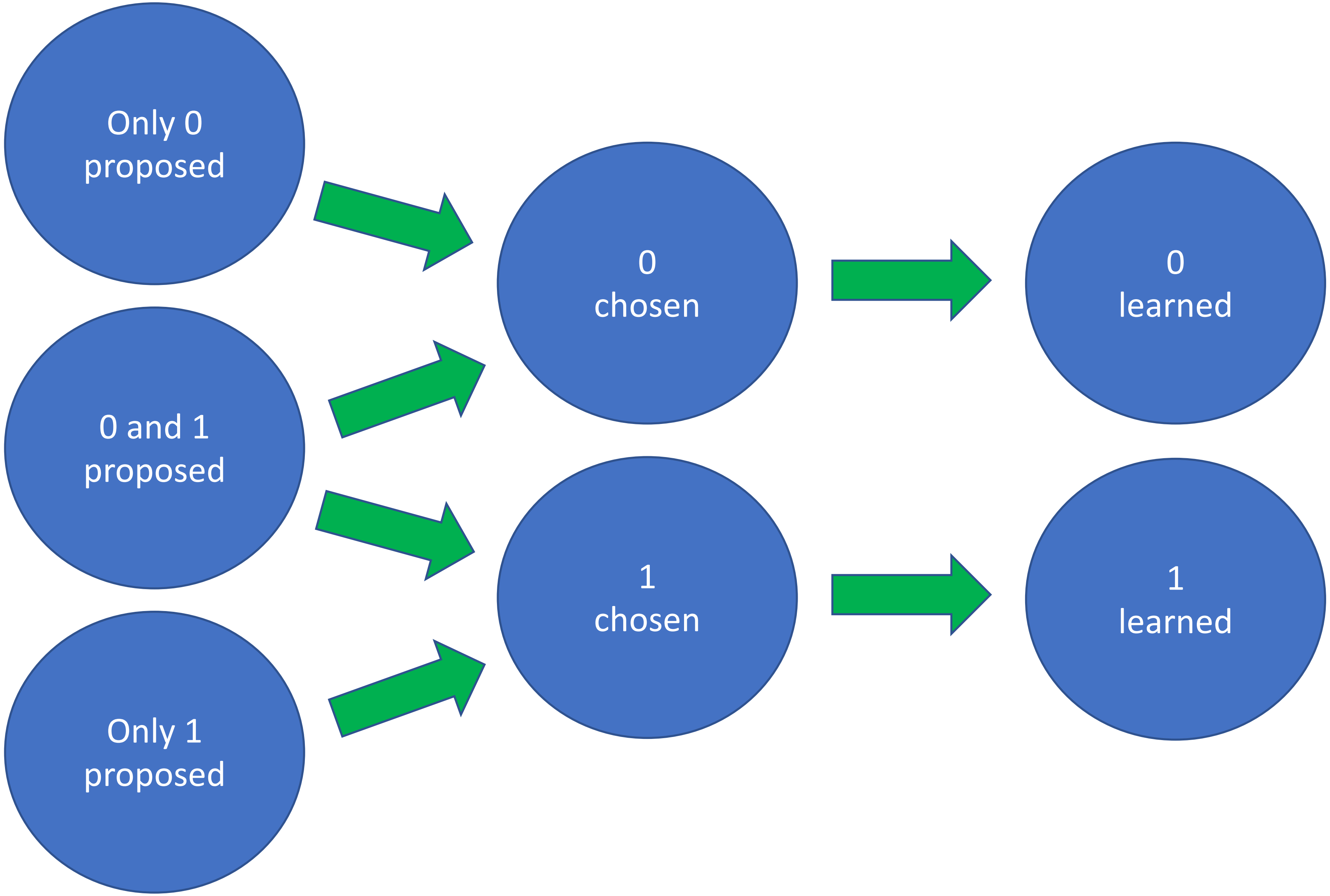
allowed by the specification

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

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”

Writing Specs

Why specify?

- To avoid errors!
  - writing a spec identifies corner cases
  - allows automated checking
    - model checking / verification
- To clarify communication between designers and builders
  - which avoids errors too…

What to specify?

- Start with the most difficult pieces
  - those pieces that are most likely to have errors in it
- Grain of atomicity
  - Too coarse may fail to reveal important details
  - Too fine may make the spec unwieldy

When to specify

- Ideally before system is implemented
  - find errors early!
- In reality, often implementation provides additional insights that may require chances to the specification
  - try to minimize this---changing the spec a lot wastes dollars and can even kill entire projects
- In practice, not unusual to write spec after an implementation is completed
  - because specs make good documentation

General hints

- Keep it simple, stupid (KISS principle)
  - spec must be clear
- Don’t be too abstract
  - may overlook details that are important in a real system
- Write comments