Prophecy made simple

Leslie Lamport and Stephan Merz

RSRS Santhosh   Amish Kansal   Kotikala Raghav

Programs for Spec A vs. Spec B

A (using num, sum)

```
def __init__(self):
    self.inp = "rdy"
    self.out = 0
    self.num = 0
    self.sum = 0

def input(self, x: int):
    assert self.inp == "rdy"
    self.inp = x
    # out, num, sum unchanged

def output(self):
    assert self.inp != "rdy"
    self.sum_0=self.sum+self.inp
    self.num_0=self.num+1
    self.out_0=self.sum_0/self.num_0
    self.inp="rdy"
```

B (using seq)

```
def __init__(self):
    self.inp = "rdy"
    self.out = 0
    self.seq = []

def input(self, x: int):
    assert self.inp == "rdy"
    self.inp = x
    self.seq.append(x)
    # out unchanged

def output(self):
    assert self.inp != "rdy"
    self.out_0=sum(self.seq)/len(self.seq)
    self.inp="rdy"
```

Preliminaries

What is a State?

- A **state** is an assignment of values to all possible variables.
- Mathematically: a mapping of the values of variables $\rightarrow$.
- Intuitively: a snapshot of the entire universe at one moment in time.
- Example: $[hr : 5]$ is a state where the clock variable $hr = 5$.

What is a Behavior?

- A **behavior** is a sequence of states:

$$s_0, s_1, s_2, \ldots$$

- Think of it as the history of system execution.
- Each adjacent pair (s_i, s_{i+1}) is a **step**.
- Example (12-hour clock):

$$[hr : 12], [hr : 1], [hr : 2], \ldots$$

- Variables not relevant to the system may take arbitrary values.

What is a Specification?

- A **specification** is a predicate on behaviors.
- It is satisfied by behaviors that represent correct system executions.
- Different from traditional verification:
  - Traditional: consider only possible executions.
  - Here: consider *all behaviors*, with irrelevant variables ignored.
- Formally, a specification in TLA has the form:

$$Init \ \land \ \Box[Next]_{\langle vars \rangle}$$

  where:
  - *Init* describes the initial state(s),
  - *Next* describes allowed steps,
  - $\Box$ means "always" (temporal operator).
- Example (hour clock):

$$(hr = 12) \ \land \ \Box\left[hr' = \text{if } hr = 12 \text{ then } 1 \text{ else } hr + 1\right]_{hr}$$

Stuttering Steps

- A **stuttering step** of a specification is a step where both states assign the same values to the specification's variables.
- Two behaviors are **stuttering-equivalent** for a specification iff they have the same sequence of non-stuttering steps.
- Often, the specification is omitted when it is clear from context.

Stuttering-Insensitive Specifications

- A specification is **stuttering-insensitive** if:

$$\sigma \text{ and } \tau \text{ are stuttering-equivalent} \quad \Rightarrow \quad (\sigma \models S \quad \Leftrightarrow \quad \tau \models S)$$

- In practice, we only write stuttering-insensitive specifications.
- This property is important for defining refinement.

Example (12-hour Clock):

$$(hr = 12) \ \wedge \ \Box\left[hr' = \begin{cases} 1 & hr = 12 \\ hr + 1 & \text{otherwise} \end{cases}\right]_{hr}$$

- Behavior A: $[hr : 12], [hr : 1], [hr : 2], [hr : 3], \ldots$
- Behavior B (with stuttering):
  $[hr : 12], [hr : 12], [hr : 1], [hr : 2], [hr : 2], [hr : 3], \ldots$
- Both satisfy the specification.

Implementation / Refinement

- We say that a specification S_1 **implements** (or **refines**) a specification S_2 iff:

$$\text{every behavior satisfying } S_1 \text{ also satisfies } S_2$$

- In temporal logic terms:

$$S_1 \text{ implements } S_2 \quad \equiv \quad \models S_1 \Rightarrow S_2$$

- i.e., the formula $S_1 \Rightarrow S_2$ is valid (true for all behaviors).

Importance for Refinement

- Refinement compares an implementation spec S_1 with an abstract spec S_2:

$$S_1 \text{ implements } S_2 \quad \Longleftrightarrow \quad \models (S_1 \Rightarrow S_2)$$

- If specs were sensitive to stuttering:
  - Extra internal steps in S_1 would break refinement.
- With stuttering-insensitivity:
  - Only **observable behavior** matters.
  - Implementations remain valid even with extra (or fewer) internal steps.

Example (12-hour clock refinement):

- Abstract spec (S_2): $[12], [1], [2], [3], \ldots$
- Implementation (S_1) with stuttering: $[12], [12], [1], [2], [2], [3], \ldots$

Both are stuttering-equivalent $\Rightarrow$ $S_1 \models S_2$.

Internal vs External Variables

- Specifications often use variables that do not represent the actual system state.
- **External variables**: describe the visible state of the system.
- **Internal variables**: auxiliary variables used to describe state changes, but not part of the observable system.
- In specifications, we want to *hide* internal variables and keep visible only external variables.

Hiding Variables in Temporal Logic

- In linear-time temporal logic, we hide a variable y in a formula F using the temporal existential quantifier $\exists$.
- Informal idea:

  $$\exists y : F \text{ is true of a behavior } \sigma \text{ iff}$$

  there exists an assignment of values to y (in each state of σ) such that the resulting behavior satisfies F.
- Problem: this definition is not stuttering-insensitive.

Correct Definition of $\exists y : F$

- Correct definition:

$$\sigma \models \exists y : F$$

  iff there exists a behavior τ such that:
  - τ is stuttering-equivalent to σ for F, and
  - there are assignments of values to y in each state of τ making $\tau \models F$.
- For a list of variables $y_1, \ldots, y_m$:

$$\exists y : F \ \triangleq \ \exists y_1 : \ldots \exists y_m : F$$

Toy Example: Counter with Buffer

External vs Internal Variables

- **External variable:** *count* — the actual counter value (observable).
- **Internal variable:** *buffer* — temporary storage, helps describe steps but not observable.

Abstract behavior (only *count*):

$$[\,count = 0\,],\ [\,count = 1\,],\ [\,count = 2\,],\ [\,count = 3\,], \ldots$$

Concrete behavior (with *buffer*):

$$[\,count = 0, buffer = 0\,],\ [\,count = 0, buffer = 1\,],\ [\,count = 1, buffer = 0\,],\ [$$

Hiding *buffer* makes the two behaviors stuttering-equivalent.

Generalized Specification

- We generalize the form of a specification S to:

$$S \triangleq \exists y : I_S$$

- where the **internal specification of S** is:

$$I_S \triangleq Init \land \Box[Next]_{\langle x,y \rangle} \land L$$

- x: list of external variables
- y: list of internal variables
- x and y are disjoint.

$\mathcal{A}$

$$
\begin{array}{lcl}
\mathcal{A} & \triangleq & \exists\, num, sum \,:\, \mathcal{IA} \\
\mathcal{IA} & \triangleq & Init_{\mathcal{A}} \land \Box[Next_{\mathcal{A}}]_{\langle in, out, num, sum \rangle} \\
Init_{\mathcal{A}} & \triangleq & (in = \mathtt{rdy}) \land (out = num = sum = 0) \\
Next_{\mathcal{A}} & \triangleq & Input_{\mathcal{A}} \lor Output_{\mathcal{A}} \\
Input_{\mathcal{A}} & \triangleq & (in = \mathtt{rdy}) \land (in' \in Int) \land \mathrm{UC}\,\langle out, num, sum \rangle \\
Output_{\mathcal{A}} & \triangleq & \quad (in \neq \mathtt{rdy}) \land (in' = \mathtt{rdy}) \\
 & & \land\ (sum' = sum + in) \land (num' = num + 1) \\
 & & \land\ (out' = sum' \,/\, num')
\end{array}
$$

B

$$
\begin{aligned}
\mathcal{B} &\triangleq \;\boldsymbol{\exists}\, seq : \mathcal{IB} \\
\mathcal{IB} &\triangleq \; Init_{\mathcal{B}} \wedge \Box[Next_{\mathcal{B}}]_{\langle in, out, seq \rangle} \\
Init_{\mathcal{B}} &\triangleq \; (in = \mathtt{rdy}) \wedge (out = 0) \wedge (seq = \langle\,\rangle) \\
Next_{\mathcal{B}} &\triangleq \; Input_{\mathcal{B}} \vee Output_{\mathcal{B}} \\
Input_{\mathcal{B}} &\triangleq \quad (in = \mathtt{rdy}) \wedge (in' \in Int) \\
&\quad\; \wedge \; (seq' = Append(seq, in')) \wedge (out' = out) \\
Output_{\mathcal{B}} &\triangleq \quad (in \neq \mathtt{rdy}) \wedge (in' = \mathtt{rdy}) \\
&\quad\; \wedge \; (out' = Sum(seq)/Len(seq)) \wedge (seq' = seq)
\end{aligned}
$$

$B \Rightarrow A$

$$\overline{num} \triangleq \textbf{if } in = rdy \textbf{ then } Len(seq) \textbf{ else } Len(Front(seq)),$$
$$\overline{sum} \triangleq \textbf{if } in = rdy \textbf{ then } Sum(seq) \textbf{ else } Sum(Front(seq)),$$

$B \Rightarrow A$

RM. If a behavior $s_1, s_2, \ldots$ satisfies $\mathcal{IB}$, then the behavior

$$s_1[\![num \leftarrow \overline{num}, sum \leftarrow \overline{sum}]\!],\ s_2[\![num \leftarrow \overline{num}, sum \leftarrow \overline{sum}]\!],\ \ldots$$

satisfies $\mathcal{IA}$.

RM1. For any state s, if s satisfies $Init_{\mathcal{B}}$, then $s[\![num \leftarrow \overline{num}, sum \leftarrow \overline{sum}]\!]$ satisfies $Init_{\mathcal{A}}$.

RM2. For any states s and t, if step s, t satisfies $Next_{\mathcal{B}} \lor \mathrm{UC}\,\langle in, out, seq\rangle$, then the pair of states

$$s[\![num \leftarrow \overline{num}, sum \leftarrow \overline{sum}]\!], t[\![num \leftarrow \overline{num}, sum \leftarrow \overline{sum}]\!]$$

satisfies $Next_{\mathcal{A}} \lor \mathrm{UC}\,\langle in, out, num, sum\rangle$.

An important Subtlety

- Not possible to prove RM2 for all pairs of states s and t.
- *Seq* could be $< rdy >$, and then *sum* can't even be defined
- Need to prove *RM*2 only for reachable states of any behaviour satisfying *IB*.

Invariants

- Find a statement that is true for all states of a behaviour satisfying *IB*.

$$\begin{aligned} Inv \;\triangleq\;& (in \in Int \cup \{\mathsf{rdy}\}) \land (out \in Int) \land (seq \in Int^*) \land \\ & ((in \neq rdy) \Rightarrow (seq \neq \langle\,\rangle) \land (in = Last(seq))), \end{aligned}$$

- Now we only need to prove:

$$\begin{aligned} & Inv \land Inv' \land Next_{\mathcal{B}} \Rightarrow \\ & \quad (Next_{\mathcal{A}} \textbf{ with } num \leftarrow \overline{num}, sum \leftarrow \overline{sum}) \lor \mathrm{UC}\,\langle in, out, \overline{num}, \overline{sum}\rangle. \end{aligned}$$

Generalization

We now generalize what we have done in this section to arbitrary specifications $\mathcal{S}_1$ and $\mathcal{S}_2$, with external variables $\mathbf{x}$, defined by

$$\begin{aligned} \mathcal{IS}_1 &\triangleq Init_1 \wedge \square[Next_1]_{\langle \mathbf{x},\mathbf{y}\rangle} \wedge L_1, \\ \mathcal{IS}_2 &\triangleq Init_2 \wedge \square[Next_2]_{\langle \mathbf{x},\mathbf{z}\rangle} \wedge L_2, \\ \mathcal{S}_1 &\triangleq \exists\!\!\exists\, \mathbf{y} : \mathcal{IS}_1 \qquad \mathcal{S}_2 \triangleq \exists\!\!\exists\, \mathbf{z} : \mathcal{IS}_2, \end{aligned} \tag{11}$$

where the lists $\mathbf{y}$ and $\mathbf{z}$ of internal variables of $\mathcal{S}_1$ and $\mathcal{S}_2$ contain no variables of $\mathbf{x}$. To verify $\mathcal{S}_1 \Rightarrow \mathcal{S}_2$, we first define a state predicate Inv, with variables in $\mathbf{x}$ and $\mathbf{y}$, and show it is an invariant of $\mathcal{IS}_1$ by showing:

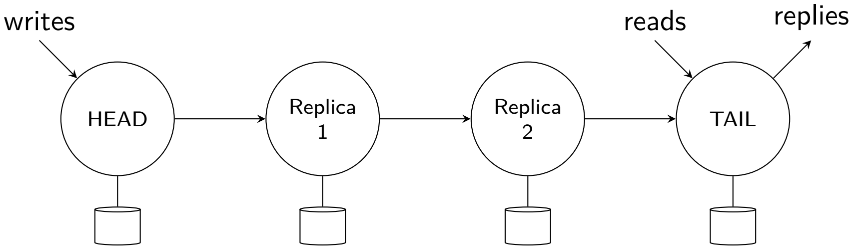
I1. $Init_1 \Rightarrow Inv$,
I2. $Inv \wedge Next_1 \Rightarrow Inv'$.

Then, if $\mathbf{z}$ is the list $z_1, \ldots, z_m$ of variables, we find expressions $\overline{z_1}, \ldots, \overline{z_m}$ with variables $\mathbf{x}$ and $\mathbf{y}$ and show the following, where $\mathbf{z} \leftarrow \overline{\mathbf{z}}$ means $z_1 \leftarrow \overline{z_1}, \ldots, z_m \leftarrow \overline{z_m}$:

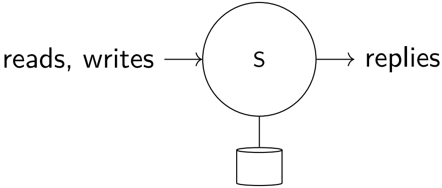
RM1. $Init_1 \Rightarrow (Init_2 \textbf{ with } \mathbf{z} \leftarrow \overline{\mathbf{z}})$,
RM2. $Inv \wedge Inv' \wedge Next_1 \Rightarrow ((Next_2 \textbf{ with } \mathbf{z} \leftarrow \overline{\mathbf{z}}) \vee \mathrm{UC}\,\langle \mathbf{x}, \overline{\mathbf{z}}\rangle)$,
RM3. $Init_1 \wedge \square[Next_1]_{\langle \mathbf{x},\mathbf{y}\rangle} \wedge L_1 \Rightarrow (L_2 \textbf{ with } \mathbf{z} \leftarrow \overline{\mathbf{z}})$.

When RM1–RM3 hold, we say that $\mathcal{IS}_1$ *implements* $\mathcal{IS}_2$ *under the refinement mapping* $\mathbf{z} \leftarrow \overline{\mathbf{z}}$.

Chain Replication is linearizable

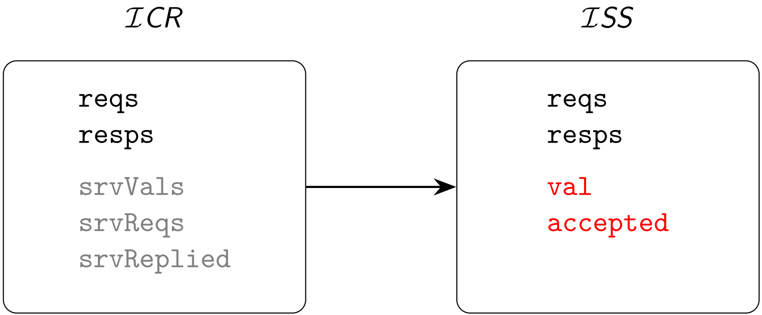


Single Server (SS)



$$
\begin{aligned}
&\text{TypeOK} \triangleq \\
&\quad \land \quad \text{reqs} \subseteq \text{Req} \\
&\quad \land \quad \text{resps} \in \text{Seq}(\text{Res}) \\
&\quad \land \quad \text{val} \in \text{VALS} \\
&\quad \land \quad \text{accepted} \in [\, \text{r} : \text{SUBSET RID},\ \text{w} : \text{SUBSET WID} \,] \\
&SS \triangleq \exists\, \text{val},\ \text{accepted} : \mathcal{I}SS
\end{aligned}
$$

$CR \Rightarrow SS$



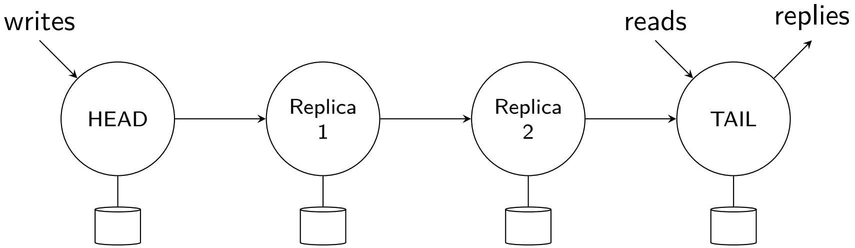
$$\mathcal{I}CR \Rightarrow (\exists\ val,\ accepted : \mathcal{I}SS)$$

$$\mathcal{I}CR \Rightarrow SS$$

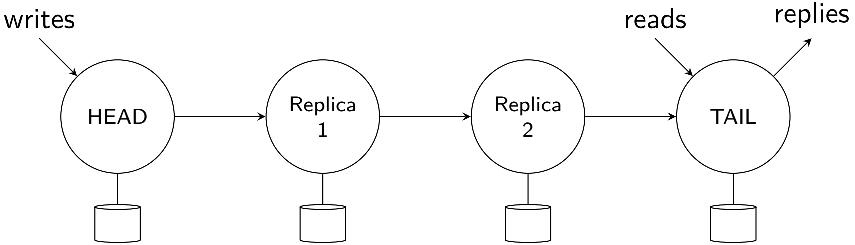
$$(\exists\ srvVals,\ srvReqs,\ srvReplied : \mathcal{I}CR) \Rightarrow SS$$

$$CR \Rightarrow SS$$

Refinement Mapping



Refinement Mapping



$$\overline{val} \leftarrow \text{srvVals}[\text{TAIL}]$$

$$\overline{accepted} \leftarrow \text{srvReplied}[\text{TAIL}]$$

Proof

$$CRInit \Rightarrow SSInit$$

$$CRNext \Rightarrow [SSNext]_{<<reqs, val, accepted, resps>>}$$

Proof

$$
\begin{array}{l}
CRInit \Rightarrow SSInit \\
\quad \land \quad reqs = \{\} \\
\quad \land \quad resps = << >> \\
\quad \land \quad srvReqs = [s \in SERVERS \rightarrow << >>] \\
\quad \land \quad srvVals = [s \in SERVERS \rightarrow INITVAL] \\
\quad \land \quad srvReplied = [s \in SERVERS \rightarrow [r \rightarrow \{\}, w \rightarrow \{\}]] \\
\Rightarrow \\
\quad \land \quad reqs = \{\} \\
\quad \land \quad resps = << >> \\
\quad \land \quad val = INITVAL \\
\quad \land \quad accepted = [r \rightarrow \{\}, w \rightarrow \{\}]
\end{array}
$$

$$CRNext \Rightarrow [SSNext]_{<<reqs, val, accepted, resps>>}$$

$$CR!IssueRead \Rightarrow SS!IssueRead$$
$$CR!IssueWrite \Rightarrow SS!IssueWrite$$

$$CR!DropRead \Rightarrow UNCHANGED << reqs, val, accepted, resps >>$$
$$CR!DropWrite \Rightarrow UNCHANGED << reqs, val, accepted, resps >>$$

$$
\begin{aligned}
&CR!ServerRead(op, s) \Rightarrow \\
&\quad IF\ s = TAIL \\
&\quad THEN\ SS!ApplyRead(op) \\
&\quad ELSE\ UNCHANGED << reqs, val, accepted, resps >>
\end{aligned}
$$

$$
\begin{aligned}
&CR!ServerWrite(op, s) \Rightarrow \\
&\quad IF\ s = TAIL \\
&\quad THEN\ SS!ApplyWrite(op) \\
&\quad ELSE\ UNCHANGED << reqs, val, accepted, resps >>
\end{aligned}
$$

Specifications A_h and B (Side by Side)

Spec A_h

$$
\begin{aligned}
A_h &\triangleq \exists num, sum, h : I_{A_h} \\
I_{A_h} &\triangleq Init_{A_h} \land \Box[Next_{A_h}]_{\{in, out, num, sum, h\}} \\
Init_{A_h} &\triangleq (in = rdy) \land (out = 0) \\
&\quad \land (num = 0) \land (sum = 0) \\
&\quad \land (h = \langle\,\rangle) \\
Next_{A_h} &\triangleq Input_{A_h} \lor Output_{A_h} \\
Input_{A_h} &\triangleq (in = rdy) \land (in' \in Int) \\
&\quad \land (h' = Append(h, in')) \\
&\quad \land (num' = num + 1) \land (sum' = sum + in') \\
&\quad \land (out' = out) \\
Output_{A_h} &\triangleq (in \neq rdy) \land (in' = rdy) \\
&\quad \land (out' = sum'/num') \land (h' = h)
\end{aligned}
$$

Spec B

$$
\begin{aligned}
B &\triangleq \exists seq : I_B \\
I_B &\triangleq Init_B \land \Box[Next_B]_{\{in, out, seq\}} \\
Init_B &\triangleq (in = rdy) \land (out = 0) \land (seq = \langle\,\rangle) \\
Next_B &\triangleq Input_B \lor Output_B \\
Input_B &\triangleq (in = rdy) \land (in' \in Int) \\
&\quad \land (seq' = Append(seq, in')) \\
&\quad \land (out' = out) \\
Output_B &\triangleq (in \neq rdy) \land (in' = rdy) \\
&\quad \land (out' = Sum(seq)/Len(seq)) \land (seq' = seq)
\end{aligned}
$$

Why $A \not\Rightarrow B$ via Refinement Mapping

- Sometimes a specification implements another, but **no refinement mapping** exists.
- In our case:
  - We proved earlier: $B \Rightarrow A$.
  - In fact, A and B are **equivalent**.
- However, $I_A \not\Rightarrow I_B$ under any refinement mapping:
  - There is **no way to define** *seq* using only the variables of A.
  - Thus, a direct mapping $A \mapsto B$ fails.
- **Solution:** Introduce an *auxiliary variable* a:

$$
\begin{aligned}
A_a &= A \text{ extended with } a, \\
A &\equiv \exists a : A_a, \\
&\text{and then show } A_a \Rightarrow B.
\end{aligned}
$$

- This construction is called **adding an auxiliary variable**.

Intuition: Why $A_h \Rightarrow B$?

Inputs: History h in A_h: Sequence *seq* i

$$in_1 \longrightarrow \langle in_1 \rangle \longrightarrow \langle in_1 \rangle$$

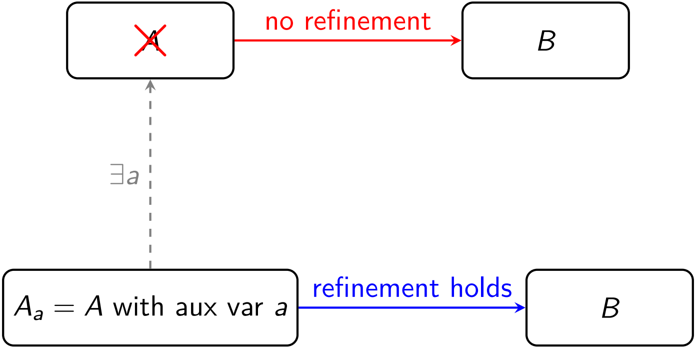
$$in_2 \longrightarrow \langle in_1, in_2 \rangle \longrightarrow \langle in_1, in_2 \rangle$$

$$in_3 \longrightarrow \langle in_1, in_2, in_3 \rangle \longrightarrow \langle in_1, in_2, in_3 \rangle$$

Output: $\frac{in_1+in_2+in_3}{3} \longrightarrow \frac{Sum(h)}{Len(h)} \longrightarrow \frac{Sum(seq)}{Len(seq)}$

Key idea: The history h in A_h evolves step by step exactly like *seq* in B. So with $seq \leftarrow h$, outputs match perfectly.

Adding Auxiliary Variable for $A \Rightarrow B$



Key idea: Direct mapping $A \mapsto B$ fails because *seq* cannot be defined in A. By adding an auxiliary variable a, we construct A_a such that $A \equiv \exists a : A_a$ and $A_a \Rightarrow B$.

Auxiliary Variables: Motivation

- Sometimes $A \Rightarrow B$ fails because B needs extra structure (e.g. sequence *seq*) not present in A.
- Fix: **Add an auxiliary variable** a to A, producing A_a.

$$A_a \quad = \quad \exists y : IS_a$$

  where

$$IS_a \quad = \quad Init_a \ \wedge\ \Box[Next_a]_{\langle x,y,a \rangle} \ \wedge\ L$$

- $Init_a$, $Next_a$ extend $Init$, $Next$ with rules for a.
- Goal: Show

$$\exists a : A_a \ \equiv\ A$$

  so that proving $A_a \Rightarrow B$ implies $A \Rightarrow B$.
- Three useful kinds of auxiliary variables:
  1. History variables (record past)
  2. Prophecy variables (predict future)
  3. Stuttering variables (insert dummy steps)

When Does A_a Really Extend A?

We must show: hiding a yields the same state machine as A.

Conditions to check

AV1: Any behavior of IS_a (with a) projects to a behavior of IS (without a). *(Soundness: no new behaviors appear once a is hidden).*

AV2: For any behavior σ of IS, we can build a behavior σ_a of IS_a by:

- inserting *stuttering steps*, and
- assigning values to a in each state.

(Completeness: no behaviors of IS are lost).

Conclusion: If AV1 and AV2 hold, then

$$\exists a : IS_a \equiv IS$$

so adding a is a legitimate auxiliary extension.

Refinement Proof: $A_h \;\Rightarrow\; B$

Goal: Show that A_h refines B under mapping $seq \leftarrow h$.

Invariant:

$$Inv : \; (num = Len(h)) \;\wedge\; (sum = Sum(h))$$

Obligations:

1. **Initial states:**

   $$Init_{A_h} \;\Rightarrow\; Init_B[seq \leftarrow h]$$

   (Both start with $in = rdy$, $out = 0$, and empty history/sequence.)

2. **Input step:**

   $$Inv \wedge Inv' \wedge Input_{A_h} \;\Rightarrow\; Input_B[seq \leftarrow h]$$

   Using $num = Len(h)$, $sum = Sum(h)$ we get preservation of h.

3. **Output step:**

   $$Inv \wedge Inv' \wedge Output_{A_h} \;\Rightarrow\; Output_B[seq \leftarrow h]$$

   From invariant: $out' = sum'/num' = Sum(h)/Len(h)$.

Refinement Proof: $A_h \;\Rightarrow\; B$ (Init & Input)

Goal: Show $A_h \Rightarrow B$ under $seq \leftarrow h$.

Invariant: $Inv : num = Len(h) \wedge sum = Sum(h)$

(a) Initial states

- $Init_{A_h} : in = rdy,\ num = 0,\ sum = 0,\ h = \langle\rangle$
- $Init_B : in = rdy,\ out = 0,\ seq = \langle\rangle$
- With $seq \leftarrow h$: both start empty $\Rightarrow$ holds ✓

(b) Input step

$$Inv \wedge Inv' \wedge Input_{A_h} \quad \Rightarrow \quad Input_B[seq \leftarrow h]$$

- $Input_{A_h} : in = rdy,\ in' \in Int,\ h' = Append(h, in'),\ num' = num + 1,\ sum' = sum + in'$
- $Input_B : in = rdy,\ in' \in Int,\ h' = Append(h, in'),\ out' = out$
- From Inv: $num = Len(h),\ sum = Sum(h)$
  $\Rightarrow num' = Len(Append(h, in')),\ sum' = Sum(Append(h, in'))$
- Invariant preserved $\Rightarrow$ Input step holds ✓

Refinement Proof: $A_h \;\Rightarrow\; B$ (Output)

Goal: Show $A_h \Rightarrow B$ under $seq \leftarrow h$.

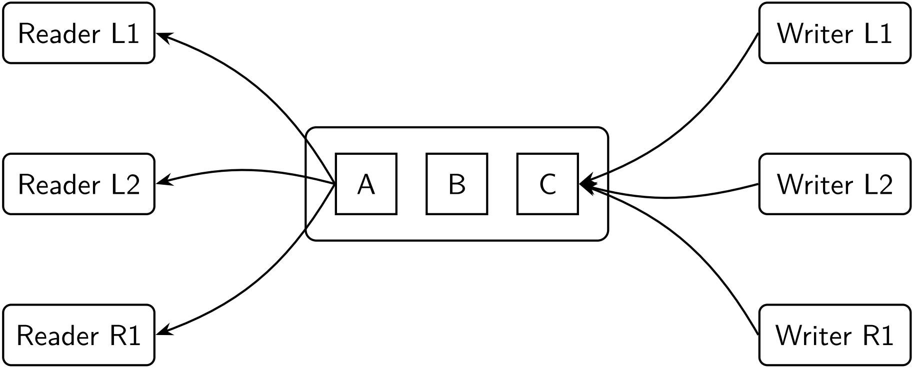
Invariant: $Inv : num = Len(h) \wedge sum = Sum(h)$

(c) Output step

$$Inv \wedge Inv' \wedge Output_{A_h} \;\;\Rightarrow\;\; Output_B[seq \leftarrow h]$$

- $Output_{A_h} : in \neq rdy,\; in' = rdy,\; out' = \frac{sum'}{num'},\; h' = h$
- $Output_B : in \neq rdy,\; in' = rdy,\; out' = \frac{Sum(h)}{Len(h)},\; h' = h$
- From Inv: $sum = Sum(h),\; num = Len(h) \Rightarrow out' = Sum(h)/Len(h)$
- Matches $Output_B \;\Rightarrow$ holds ✓

Linearizable queue



Queue

$$CONSTANTS\ Values, none$$

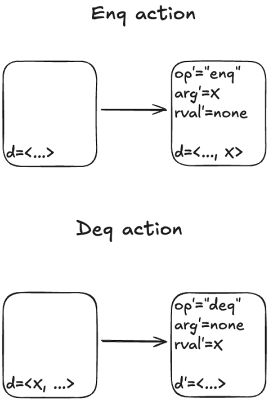
$$QTypeOK \triangleq$$

$$\begin{aligned}
&\land\quad op \in \{"enq", "deq", ""\}\\
&\land\quad arg \in Values \cup \{none\}\\
&\land\quad rval \in Values \cup \{none\}\\
&\land\quad d \in Seq(Values)
\end{aligned}$$

$$QInit \triangleq$$

$$\begin{aligned}
&\land\quad op = ""\\
&\land\quad arg = none\\
&\land\quad rval = none\\
&\land\quad d = << >>
\end{aligned}$$

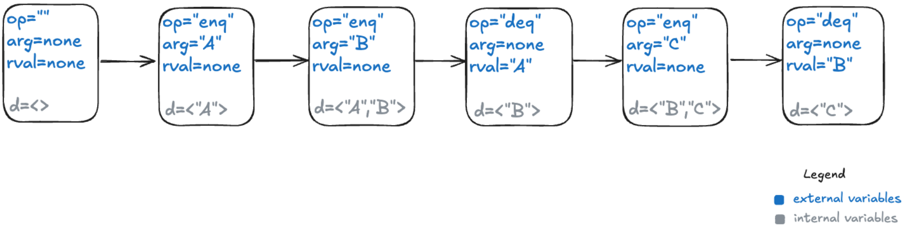
Queue



$$
\begin{aligned}
&Enq \triangleq \\
&\exists\, x \in Values : \\
&\quad \land \quad op' = \text{"}enq\text{"} \\
&\quad \land \quad arg' = x \\
&\quad \land \quad rval' = none \\
&\quad \land \quad d' = Append(d, x)
\end{aligned}
$$

$$
\begin{aligned}
&Deq \triangleq \\
&\quad \land \quad d \neq <<>> \\
&\quad \land \quad op' = \text{"}deq\text{"} \\
&\quad \land \quad arg' = none \\
&\quad \land \quad rval' = Head(d) \\
&\quad \land \quad d' = Tail(d)
\end{aligned}
$$

Queue


op=""
arg=none
rval=none
d=<>
op="enq"
arg="A"
rval=none
d=<"A">
op="enq"
arg="B"
rval=none
d=<"A","B">
op="deq"
arg=none
rval="A"
d=<"B">
op="enq"
arg="C"
rval=none
d=<"B","C">
op="deq"
arg=none
rval="B"
d=<"C">
Legend
external variables
internal variables

Linearizable Queue

$CONSTANTS\ Values, Threads, none$

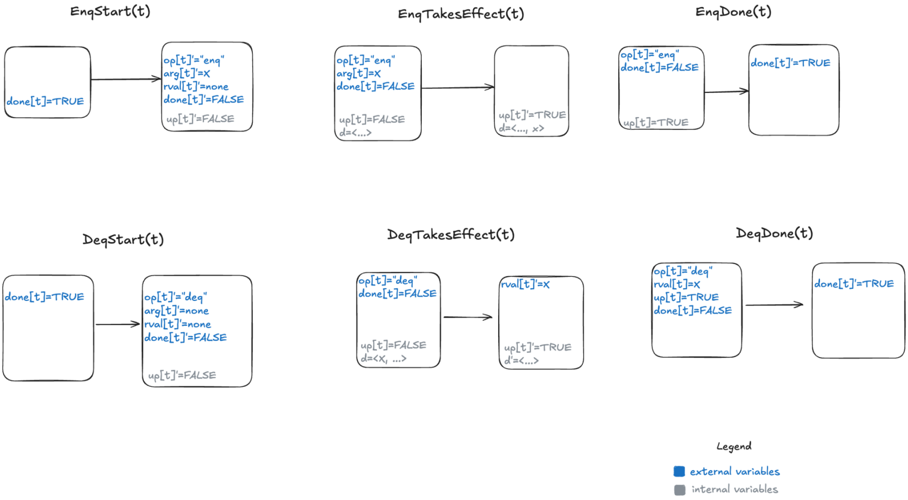
$LQTypeOK \triangleq$

$$
\begin{aligned}
\land\ & op \in [Threads \rightarrow \{"enq", "deq", ""\}] \\
\land\ & arg \in [Threads \rightarrow Values \cup \{none\}] \\
\land\ & rval \in [Threads \rightarrow Values \cup \{none\}] \\
\land\ & done \in [Threads \rightarrow BOOLEAN] \\
\land\ & up \in [Threads \rightarrow BOOLEAN] \\
\land\ & d \in Seq(Values)
\end{aligned}
$$

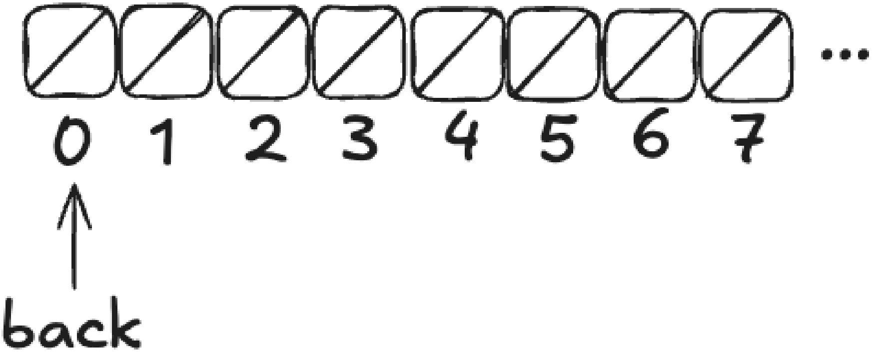
$LQInit \triangleq$

$$
\begin{aligned}
\land\ & op = [Threads \rightarrow ""] \\
\land\ & arg = [Threads \rightarrow none] \\
\land\ & rval = [Threads \rightarrow none] \\
\land\ & done = [Threads \rightarrow TRUE] \\
\land\ & up = [Threads \rightarrow TRUE] \\
\land\ & d = << >>
\end{aligned}
$$

Linearizable Queue

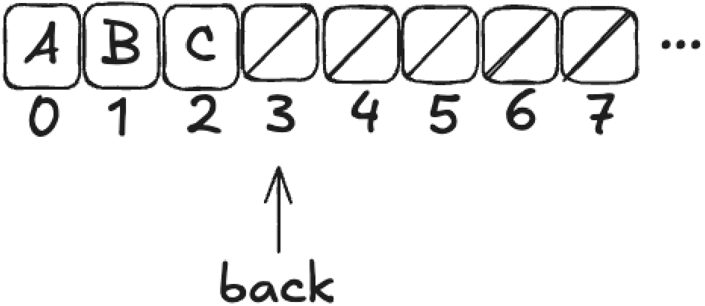


Herlihy & Wing Queue



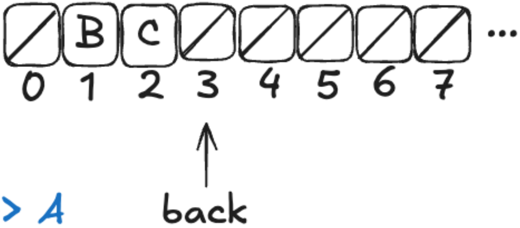
Enqueue

```
Enq = proc (q: queue, x: item)
  i: int := INC(q.back)    % Allocate a new slot.
  STORE (q.items[i], x)    % Fill it.
  end Enq
```



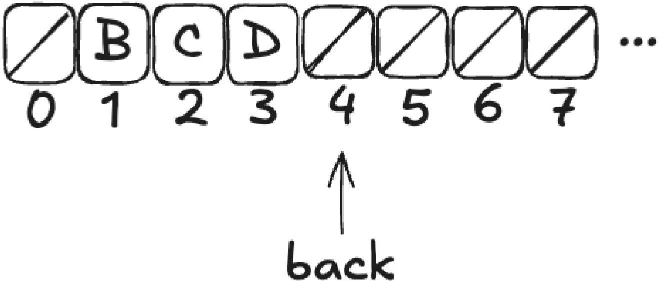
Dequeue

```
Deq = proc (q: queue) returns (item)
  while true do
    range: int := READ(q.back) - 1
    for i: int in 1 .. range do
      x: item := SWAP(q.items[i], null)
      if x ~= null then return(x) end
      end
    end
  end Deq
```

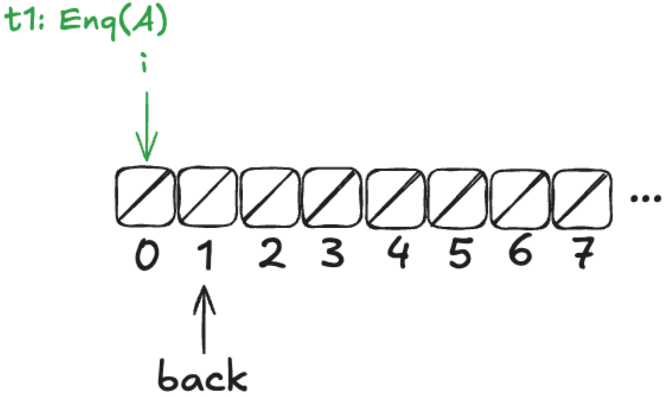


Deq() -> A

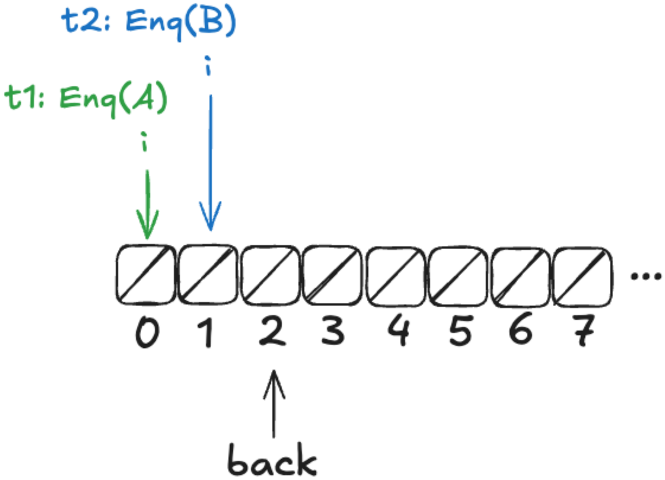
Enqueue after Dequeue



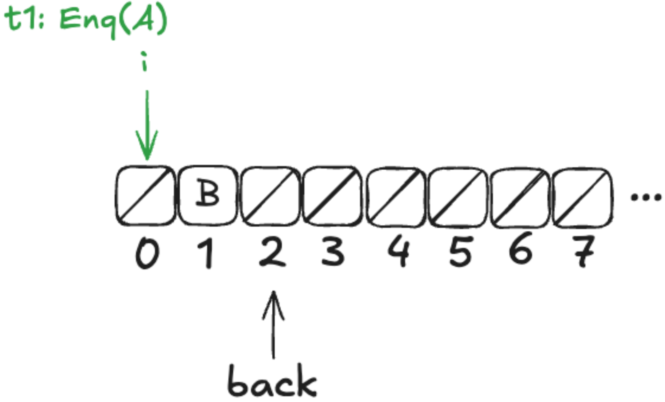
Refinement Challenge



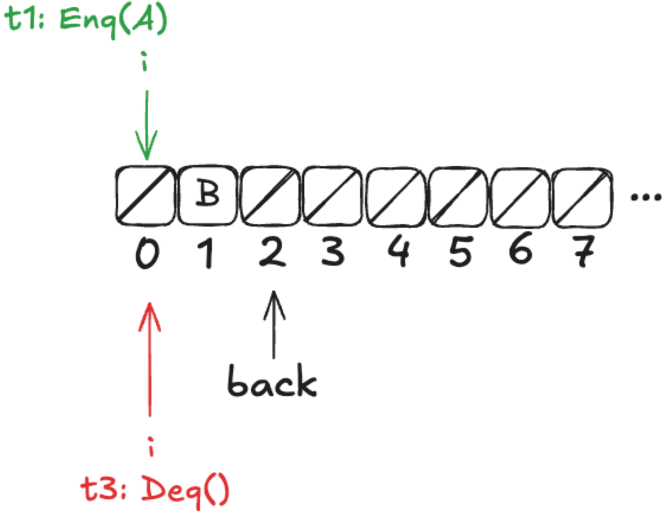
Refinement Challenge



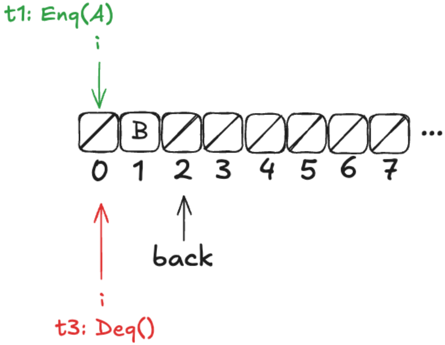
Refinement Challenge



Refinement Challenge

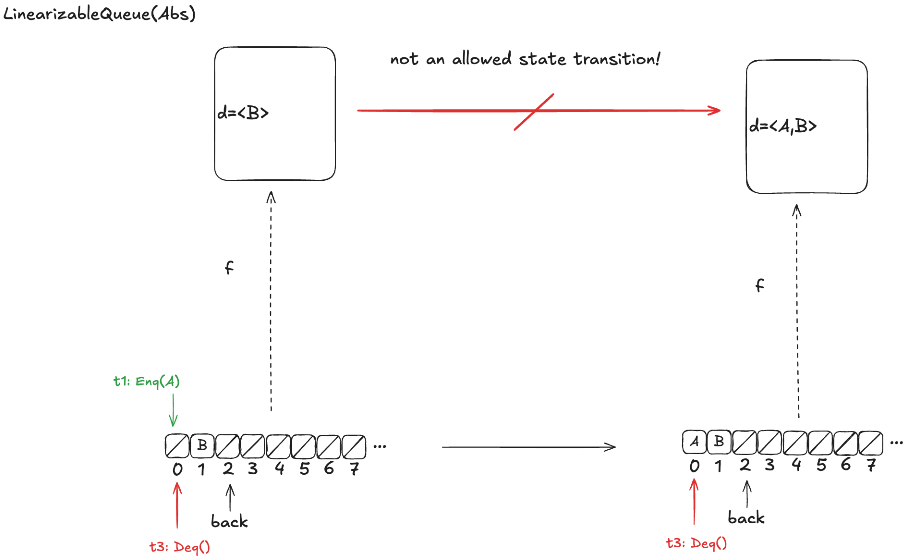


Possibilities

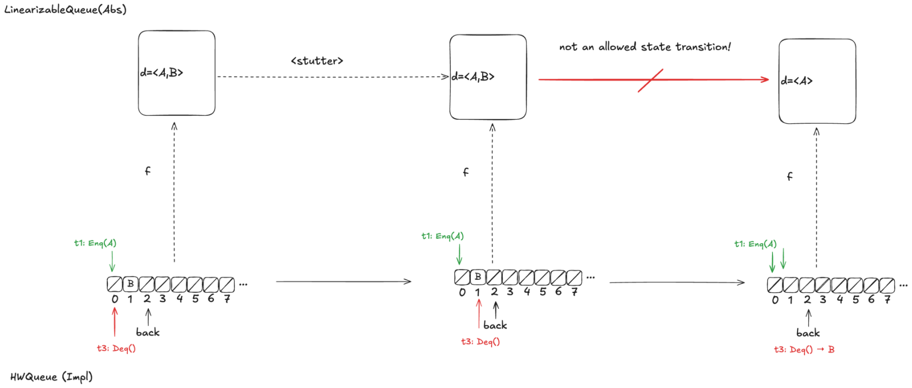


1. $d = < B >$
2. $d = < A, B >$
3. $d = < B, A >$

Option 1: $d =< B >$

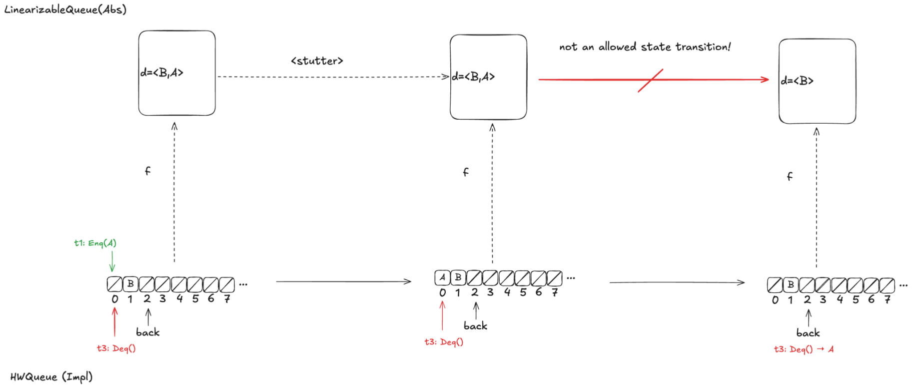

LinearizableQueue(Abs)
not an allowed state transition!
d=<B>
d=<A,B>
f
f
t1: Enq(A)
B
0 1 2 3 4 5 6 7
back
t3: Deq()
A B
0 1 2 3 4 5 6 7
back
t3: Deq()

Option 2: $d =< A, B >$



LinearizableQueue(Abs)
d=<A,B>
<stutter>
d=<A,B>
not an allowed state transition!
d=<A>
f
f
f
t1: Enq(A)
B
0 1 2 3 4 5 6 7 ...
back
t3: Deq()
t1: Enq(A)
B
0 1 2 3 4 5 6 7 ...
back
t3: Deq()
t1: Enq(A)
0 1 2 3 4 5 6 7 ...
back
t3: Deq() → B
HWQueue (Impl)

Option 3: $d =< B, A >$


LinearizableQueue(Abs)
d=<B,A>
<stutter>
d=<B,A>
not an allowed state transition!
d=<B>
f
f
f
t1: Enq(A)
B
0 1 2 3 4 5 6 7
back
t3: Deq()
A B
0 1 2 3 4 5 6 7
back
t3: Deq()
B
0 1 2 3 4 5 6 7
back
t3: Deq() → A
HWQueue (Impl)

Toy Example

- We demonstrate the problem again with a toy example
- Will discuss how to solve it using prophecy variables

A

$$A \triangleq \exists in, num, sum : IA$$

$$\begin{array}{lcl}
\mathcal{IA} & \triangleq & Init_{\mathcal{A}} \land \Box[Next_{\mathcal{A}}]_{\langle in, out, num, sum \rangle} \\
Init_{\mathcal{A}} & \triangleq & (in = \mathsf{rdy}) \land (out = num = sum = 0) \\
Next_{\mathcal{A}} & \triangleq & Input_{\mathcal{A}} \lor Output_{\mathcal{A}} \\
Input_{\mathcal{A}} & \triangleq & (in = \mathsf{rdy}) \land (in' \in Int) \land \mathrm{UC}\,\langle out, num, sum \rangle \\
Output_{\mathcal{A}} & \triangleq & \quad (in \neq \mathsf{rdy}) \land (in' = \mathsf{rdy}) \\
 & & \land\ (sum' = sum + in) \land (num' = num + 1) \\
 & & \land\ (out' = sum' \,/\, num')
\end{array}$$

C

$$
\begin{aligned}
C &\triangleq \boldsymbol{\exists}\, in, num, sum : \mathit{IC} \\
\mathit{IC} &\triangleq Init_{\mathcal{A}} \land \Box[Next_C]_{\langle out, in, num, sum \rangle} \\
Next_C &\triangleq Next_{\mathcal{A}} \lor Undo_C \\
Undo_C &\triangleq (in \neq \mathsf{rdy}) \land (in' = \mathsf{rdy}) \land \mathrm{UC}\,\langle out, num, sum\rangle
\end{aligned}
$$

C^p

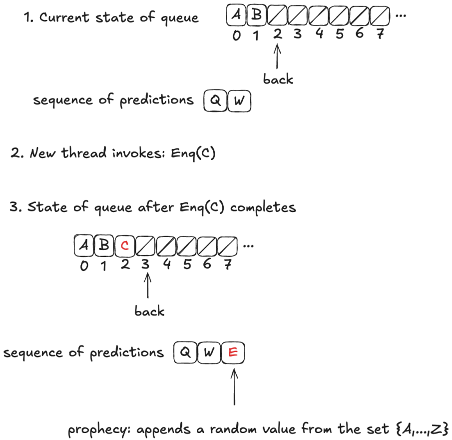
$$
\begin{array}{lcl}
C^p & \triangleq & \exists\, in, num, sum \;:\; \mathcal{I}C^p \\
\mathcal{I}C^p & \triangleq & Init^p_C \;\wedge\; \Box[Next^p_C]_{\langle out, in, num, sum, p \rangle} \\
Init^p_C & \triangleq & (p \in \{\mathsf{do}, \mathsf{undo}\}) \;\wedge\; Init_{\mathcal{A}} \\
Next^p_C & \triangleq & Input^p_{\mathcal{A}} \;\vee\; Output^p_{\mathcal{A}} \;\vee\; Undo^p_C \\
Input^p_C & \triangleq & (p' = p) \;\wedge\; Input_{\mathcal{A}} \\
Output^p_C & \triangleq & (p = \mathsf{do}) \;\wedge\; (p' \in \{\mathsf{do}, \mathsf{undo}\}) \wedge Output_{\mathcal{A}} \\
Undo^p_C & \triangleq & (p = \mathsf{undo}) \;\wedge\; (p' \in \{\mathsf{do}, \mathsf{undo}\}) \wedge Undo_C
\end{array}
$$

$C^P \Rightarrow A$

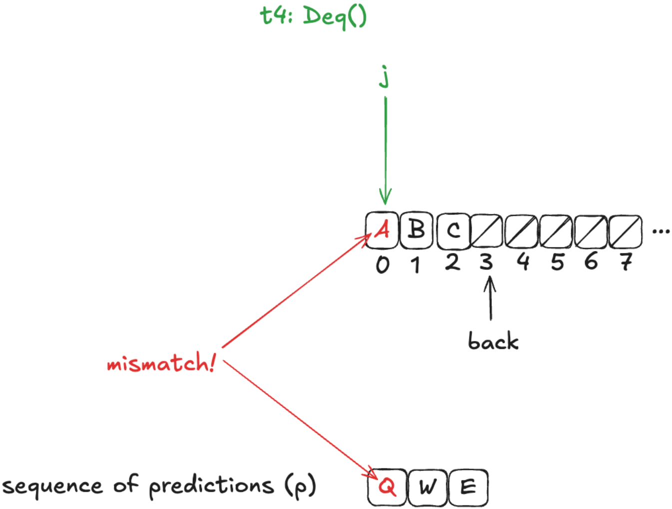
$C^P \Rightarrow A$ under the following refinement mapping.

$$in \leftarrow \textbf{if } p = \textsf{undo} \textbf{ then } \textsf{rdy} \textbf{ else } in, \quad num \leftarrow num, \quad sum \leftarrow sum.$$

Prophecy


1. Current state of queue
A B ...
0 1 2 3 4 5 6 7
back
sequence of predictions Q W
2. New thread invokes: Enq(C)
3. State of queue after Enq(C) completes
A B C ...
0 1 2 3 4 5 6 7
back
sequence of predictions Q W E
prophecy: appends a random value from the set {A,...,Z}

Disallow violating transitions



Acknowledgments

Illustrations for Linearizable Queue & Herlihy Wing Queue were adapted from this article from surfing complexity by Lorin Hochstein

The TLA+ specs for linearizable queue, H&W queue and H&W queue with prophecy can be found here.