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: $[var_1 = value_1, var_2 = value_2, \ldots]$
  • Meaning: a state in which $var_1$ has value $value_1$, …
  • 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. $[hr = 3. temp = 62], [hr = 3. temp = 68], \ldots$
  • 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: $HC_{ini} \triangleq hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
  - Or, informally, $HC_{ini} \triangleq hr \in \{1, \cdots, 12\}$
  - $HC_{ini}$ is simply a name given to the predicate

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

- 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
  • \( \{(state_1^{\text{pre}}, state_1^{\text{post}}), (state_2^{\text{pre}}, state_2^{\text{post}}), \cdots \} \)

• Example:
  • \( \text{HCnxt} \triangleq \{ ([hr = 11], [hr = 12]), ([hr = 12], [hr = 1]), \cdots \} \)
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: $HC_{nxt} \triangleq hr' = hr \mod 12 + 1$ (% is the “modulo” operator)
  - or, $HC_{nxt_2} \triangleq hr' = \text{IF } hr = 12 \text{ THEN } 1 \text{ ELSE } hr + 1$
  - But note that $HC_{nxt_2} \not\equiv HC_{nxt}$
- $hr'$ 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: $HC_{nxt} \triangleq hr' - hr = 1 \mod 12$
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 \mod 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: \([hr=3. \ temp = 62]\) → \([hr = 3. \ temp = 63]\)
- This is called a *stuttering step* of the clock
  - i.e., \(hr' = hr\)
Final specification: hardware clock

Module HourClock
• Variable \( hr \)
• \( HCini \triangleq hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 \} \)
• \( HCnxt \triangleq hr' = hr \mod 12 + 1 \)
• \( HC \triangleq HCini \land \square (HCnxt \lor (hr' = hr)) \)

The latter can be abbreviated using the following TLA+ notation

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

([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, \( HC \Rightarrow \Box HC_{ini} \)
- Proof: by induction on \#steps
A note on variables and types

• Variables in TLA+ are untyped

• However, if one can prove $\text{SPEC } \Rightarrow \Box \; \nu \in S$ for some variable $\nu$ and constant set $S$, then one can call $S$ the type of $\nu$ 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: $\text{HCtypeInvariant } \triangleq hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$

• Note, in this case $\text{HCtypeInvariant } \equiv \text{HCini}$
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

\[ isPrime(q) \equiv q > 1 \land \forall r \in 2 \ldots (q - 1) : q \% r \neq 0 \]

TypeInvariant \( \equiv \) isPrime(p)

Init \( \equiv \) p = 2

Next \( \equiv \) p' > p \land isPrime(p') \land \forall q \in (p + 1) \ldots (p' - 1) : \neg isPrime(q)

Spec \( \equiv \) Init \land \Box[Next]_p

THEOREM Spec \( \Rightarrow \Box TypeInvariant \)
Spec that generates all prime numbers

-------------------------------- M O D U L E  p r i m e  --------------------------------
EXTENDS Naturals
VARIABLE p

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

TypeInvariant == isPrime(p)

Init == p = 2
Next == p' > p \land isPrime(p') \land \forall q \in (p+1)..(p'-1): \neg isPrime(q)

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

THEOREM Spec => []TypeInvariant
Asynchronous FIFO Channel Specification

\[
\begin{align*}
Send & \triangleq \land rdy = \text{ack} \\
& \land \text{val'} \in \text{Data} \\
& \land rdy' = 1 - rdy \\
& \land \text{ack'} = \text{ack}
\end{align*}
\]

\[
\begin{align*}
Recv & \triangleq \land rdy \neq \text{ack} \\
& \land \text{ack'} = 1 - \text{ack} \\
& \land \text{val'} = \text{val} \\
& \land rdy' = rdy
\end{align*}
\]
Asynchronous FIFO Channel Specification

**TypeInvariant** ≜ \( \land \ val \in Data \)
\( \land \ rdy \in \{0, 1\} \)
\( \land \ ack \in \{0, 1\} \)

**Init** ≜ \( \land \ val \in Data \)
\( \land \ rdy \in \{0, 1\} \)
\( \land \ ack = rdy \)

**Send** ≜ \( \land \ rdy = ack \)
\( \land \ val' \in Data \)
\( \land \ rdy' = 1 - rdy \)
\( \land \ ack' = ack \)

**Recv** ≜ \( \land \ rdy \neq ack \)
\( \land \ ack' = 1 - ack \)
\( \land \ val' = val \)
\( \land \ rdy' = rdy \)

**Next** ≜ **Send** \( \lor \) **Recv**

**Spec** ≜ **Init** \( \land \ \Box[\text{Next}]\langle rdy, ack, val \rangle \)
Asynchronous FIFO Channel Specification
introducing operators with arguments

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

Next $\triangleq \lor \text{Send}$
$\lor \text{Recv}$

Send($d$) $\triangleq \land rdy = \text{ack}$
$\land \text{val'} = d$
$\land rdy' = 1 - rdy$
$\land \text{ack'} = \text{ack}$

Next $\triangleq \lor \exists d \in Data: \text{Send}(d)$
$\lor \text{Recv}$
Asynchronous FIFO Channel Specification
introducing records

TypeInvariant ≜ chan ∈ [val: Data, rdy: {0,1}, ack: {0,1}]

Init ≜ chan.val ∈ Data ∧ chan.rdy ∈ { 0, 1 } ∧ chan.ack = chan.rdy

Send(d) ≜ chan.rdy = chan.ack ∧ chan’ =
    [ val ↦ d, rdy ↦ 1 − chan.rdy, ack ↦ chan.ack ]

Recv ≜ chan.rdy ≠ chan.ack ∧ chan’ =
    [ val ↦ chan.val, rdy ↦ chan.rdy, ack ↦ 1 − chan.ack ]

Next ≜ ∃d ∈ Data: Send(d) ∨ Recv

Spec ≜ Init ∧ □[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 $\sigma$
• $\sigma \models F$ means that $F$ holds over $\sigma$
• $F$ is a theorem if $\sigma \models F$ holds over all behaviors $\sigma$
• 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 $\sigma$
  • i.e., the first step in $\sigma$ is an $A$ step
• 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 $\sigma \models \Box F \equiv \forall n \in \mathbb{N}: \sigma^{+n} \models F$
Boolean combinations of temporal formulas

• $\sigma \models (F \land G) \triangleq (\sigma \models F) \land (\sigma \models G)$
• $\sigma \models (F \lor G) \triangleq (\sigma \models F) \lor (\sigma \models G)$
• $\sigma \models \neg F \triangleq \neg (\sigma \models F)$
• $\sigma \models (F \Rightarrow G) \triangleq (\sigma \models F) \Rightarrow (\sigma \models G)$
• $\sigma \models (\exists r: F) \triangleq \exists r: \sigma \models F$
• $\sigma \models (\forall r \in S: F) \triangleq \forall r \in S: \sigma \models F$  \hspace{1cm}  // if $S$ is a constant set
Example

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

$\sigma \vdash \Box((x = 1) \Rightarrow \Box(y > 0))$

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

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

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

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 \([x = 1] \rightarrow [x = 1] \rightarrow [x = 2]\)
  • \([x' = x + 1 ]_x\) satisfied by \([x = 1] \rightarrow [x = 1] \rightarrow [x = 3]\)
    but not by \([x = 1] \rightarrow [x = 3]\)

• Yet \( \Box[x' = x + 1 ]_x \) is a TLA+ formula!
Eventually $F$

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

$\sigma \models \Diamond F$

$\equiv \sigma \models \neg \Box \neg F$

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

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

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

$\equiv \exists n \in \mathbb{N}: (\sigma^{+n} \models F)$
Eventually an $A$ step occurs that changes $v$...

\[ \Diamond \langle A \rangle_v \triangleq \neg \square [\neg A]_v \]

\[ \sigma \models \Diamond \langle A \rangle_v \]
\[ \equiv \sigma \models \neg \square [\neg A]_v \]
\[ \equiv \neg (\sigma \models \square [\neg A]_v) \]
\[ \equiv \neg (\forall n \in \mathbb{N}: \sigma^+ n \models [\neg A]_v) \]
\[ \equiv \neg (\forall n \in \mathbb{N}: \sigma^+ n \models (\neg A \lor v' = v)) \]
\[ \equiv \exists n \in \mathbb{N}: \sigma^+ n \models (A \land v' \neq v) \]
HourClock revisited

Module HourClock

Variable \( hr \)

- \( \text{HCini} \triangleq hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\} \)
- \( \text{HCnxt} \triangleq hr' = hr \mod 12 + 1 \)
- \( \text{HC} \triangleq \text{HCini} \land [\Box[\text{HCnxt}]_{hr} \land [\text{HCnxt}]_{hr}] \)

\textit{hr is a parameter of the specification HourClock}
HourClock with *liveness* clock that never stops

Module HourClock

Variable *hr*

- $\text{HCini} \triangleq hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
- $\text{HCnxt} \triangleq hr' = hr \mod 12 + 1$
- $\text{HC} \triangleq \text{HCini} \land \Box[H\text{Cnxt}]_{hr}$
- $\text{LiveHC} \triangleq \text{HC} \land \Box(\Diamond(H\text{Cnxt})_{hr})$
Module Channel with Liveness

Constant Data

Variable chan

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

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

Send(d) \triangleq \text{chan.rdy} = \text{chan.ack} \land \text{chan'} =

\begin{align*}
\begin{array}{l}
[ \text{val} \leftrightarrow d, \text{rdy} \leftrightarrow 1 - \text{chan.rdy}, \text{ack} \leftrightarrow \text{chan.ack} ]
\end{array}
\end{align*}

Recv \triangleq \text{chan.rdy} \neq \text{chan.ack} \land \text{chan'} =

\begin{align*}
\begin{array}{l}
[ \text{val} \leftrightarrow \text{chan.val}, \text{rdy} \leftrightarrow \text{chan.rdy}, \text{ack} \leftrightarrow 1 - \text{chan.ack} ]
\end{array}
\end{align*}

Next \triangleq \exists d \in \text{Data}: \text{Send}(d) \lor \text{Recv}

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

LiveSpec \triangleq \text{Spec} \land \square (\Diamond \langle \text{Next} \rangle_{\text{chan}})
Module Channel with Liveness

Constant $Data$  

Variable $chan$

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

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

$Send(d) \triangleq chan.\text{rdy} = chan.\text{ack} \land chan' =$

$[ val \leftrightarrow d, rdy \leftrightarrow 1 - chan.\text{rdy}, ack \leftrightarrow chan.\text{ack} ]$

$Recv \triangleq chan.\text{rdy} \neq chan.\text{ack} \land chan' =$

$[ val \leftrightarrow chan.\text{val}, rdy \leftrightarrow 1 - chan.\text{rdy}, ack \leftrightarrow chan.\text{ack} ]$

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

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

$LiveSpec \triangleq Spec \land \Box(\Diamond\langle Next\rangle_{chan})$
Module Channel with Liveness

Constant \textit{Data} \quad \text{Variable} \textit{chan}

\text{TypeInvariant} \triangleq \textit{chan} \in [\text{val}\colon \text{Data}, \text{rdy}\colon\{0,1\}, \text{ack}\colon\{0,1\}]

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

\text{Send}(d) \triangleq \textit{chan}.\text{rdy} = \textit{chan}.\text{ack} \land \textit{chan}^{'} =
\begin{align*}
&[\text{val} \mapsto d, \text{rdy} \mapsto 1 - \textit{chan}.\text{rdy}, \text{ack} \mapsto \textit{chan}.\text{ack}]
\end{align*}

\text{Recv} \triangleq \textit{chan}.\text{rdy} \neq \textit{chan}.\text{ack} \land \textit{chan}^{'} =
\begin{align*}
&[\text{val} \mapsto \textit{chan}.\text{val}, \text{rdy} \mapsto \textit{chan}.\text{rdy}, \text{ack} \mapsto 1 - \textit{chan}.\text{ack}]
\end{align*}

\text{Next} \triangleq \exists d \in \text{Data}: \text{Send}(d) \lor \text{Recv}

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

\text{LiveSpec} \triangleq \text{Spec} \land \square(\textit{chan}.\text{rdy} \neq \textit{chan}.\text{ack} \Rightarrow \Diamond\langle\text{Recv}\rangle_{\text{chan}})$
Weak Fairness as a liveness condition

- \textbf{ENABLED} \langle A \rangle_v \text{ means action } A \text{ is possible in some state}
  - State predicate conjuncts all hold \textit{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: \( WF_{hr}(HC_{nxt}) \)
- Channel: \( WF_{chan}(Recv) \)
(surprising) Weak Fairness equivalence

\[WF_v(A) \triangleq \Box(\Box\text{ENABLED } \langle A \rangle_v \Rightarrow \Diamond \langle A \rangle_v)\]

\[\equiv \Box\Diamond (\neg \text{ENABLED } \langle A \rangle_v) \lor \Box\Diamond \langle A \rangle_v\]

\[\equiv \Diamond \Box (\text{ENABLED } \langle A \rangle_v) \Rightarrow \Box\Diamond \langle A \rangle_v\]

• 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

\[ SF_v(A) \equiv \Diamond \Box (\neg \text{ENABLED } \langle A \rangle_v) \lor \Box \Diamond \langle A \rangle_v \]
\[ \equiv \Box \Diamond (\text{ENABLED } \langle A \rangle_v) \Rightarrow \Box \Diamond \langle A \rangle_v \]

- \( A \) 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

\textit{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** *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' =  

\[ [ \text{val} \leftrightarrow d, \text{rdy} \leftrightarrow 1 - \text{chan}.\text{rdy}, \text{ack} \leftrightarrow \text{chan}.\text{ack} ] \]

Recv $\triangleq$ chan.rdy $\neq$ chan.ack $\land$ chan' =  

\[ [ \text{val} \leftrightarrow \text{chan}.\text{val}, \text{rdy} \leftrightarrow \text{chan}.\text{rdy}, \text{ack} \leftrightarrow 1 - \text{chan}.\text{ack} ] \]

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

Spec $\triangleq$ Init $\land$ $\Box[\text{Next}]_{chan}$
Instantiating a Channel

\[
\text{InChan} \triangleq \text{INSTANCE Channel WITH Data } \leftarrow \text{Message, chan } \leftarrow \text{in}
\]

TypeInvariant \triangleq \text{chan } \in \{\text{val: Data, rdy: \{0,1\}, ack: \{0,1\}}\}

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

*Instantiation is Substitution!*
MODULE InnerFIFO

EXTENDS Naturals, Sequences

CONSTANT Message

VARIABLES in, out, q

InChan ≜ INSTANCE Channel WITH Data ← Message, chan ← in

OutChan ≜ INSTANCE Channel WITH Data ← Message, chan ← out

Init ≜ ∧ InChan!Init ∧ OutChan!Init ∧ q = ⟨⟩

TypeInvariant ≜ ∧ InChan!TypeInvariant ∧ OutChan!TypeInvariant ∧ q ∈ Seq(Message)
\[ SSend(msg) \triangleq \wedge InChan!Send(msg) \wedge UNCHANGED \langle out, q \rangle \]

Send \( msg \) on channel \( \text{in} \).

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

Receive message from channel \( \text{in} \) and append it to tail of \( q \).

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

Enabled only if \( q \) is nonempty. Send \( \text{Head}(q) \) on channel \( \text{out} \) and remove it from \( q \).

\[ RRcv \triangleq \wedge OutChan!Rcv \wedge UNCHANGED \langle in, q \rangle \]

Receive message from channel \( \text{out} \).
\[\begin{align*}
\text{Next} & \triangleq \forall \exists \text{msg} \in \text{Message} : \text{SSend}(\text{msg}) \\
& \quad \lor \text{BufRcv} \\
& \quad \lor \text{BufSend} \\
& \quad \lor \text{RRcv} \\
\text{Spec} & \triangleq \text{Init} \land \Box [\text{Next}]_{i, o, q}
\end{align*}\]

THEOREM  \[\text{Spec} \Rightarrow \Box \text{TypeInvariant}\]
Parametrized Instantiation

\[
InChan \triangleq \text{INSTANCE Channel WITH Data } \leftarrow \text{Message}, \text{chan } \leftarrow i
\]

\[
Chan(ch) \triangleq \text{INSTANCE Channel WITH Data } \leftarrow \text{Message}, \text{chan } \leftarrow ch
\]

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

\[
Chan(in)!\text{TypeInvariant } \equiv \text{in } \in [\text{val: Message, rdy: } \{0,1\}, \text{ack: } \{0,1\}]
\]
Internal (= Non-Interface) Variables

There is no $q$ here

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

$Inner(q) \triangleq \text{INSTANCE InnerFIFO}$

$Spec \triangleq \exists q : Inner(q)!Spec$
Hiding Internal Variables

MODULE FIFO

CONSTANT Message
VARIABLES in, out

\[ Inner(q) \triangleq \text{INSTANCE InnerFIFO} \]
\[ Spec \triangleq \exists q : Inner(q)!Spec \]

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 implement a bounded buffer
• That is, $\square \text{len}(q) \leq N$ for some constant $N > 0$
• The only place where $q$ is extended is in $BufRcv$

\[
BufRcv \triangleq \ \land \ \text{InChan}!Rcv \\
\land \ q' = \text{Append}(q, \text{in.val}) \\
\land \ \text{UNCHANGED} \ \text{out}
\]
Pretty. Now for something cool!

• Suppose we wanted to implemented a bounded buffer
• That is, $\square \text{len}(q) \leq N$ for some constant $N > 0$
• The only place where $q$ is extended is in $\text{BufRcv}$

\[
\text{BufRcv} \overset{\Delta}{=} \quad \land \text{InChan}!\text{Rcv} \\
\land q' = \text{Append}(q, \text{in.val}) \\
\land \text{UNCHANGED out} \\
\land \text{len}(q) < N
\]
Even cooler (but tricky)

**MODULE BoundedFIFO**

EXTENDS Naturals, Sequences
VARIABLES in, out
CONSTANT Message, N

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

\[ \text{Inner}(q) \triangleq \text{INSTANCE InnerFIFO} \]

\[ B\text{Next}(q) \triangleq \land \text{Inner}(q)!\text{Next} \land \text{Inner}(q)!\text{BufRcv} \Rightarrow (\text{Len}(q) < N) \]

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

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 $(N \in \text{Nat}) \land (N > 0)$

\[
\text{Inner}(q) \triangleq \text{INSTANCE InnerFIFO}
\]

\[
\text{BNext}(q) \triangleq \land \text{Inner}(q)!\text{Next} \\
\land \text{Inner}(q)!\text{BufRecv} \Rightarrow (\text{Len}(q) < N)
\]

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

Based on material from Section 10.8, Specifying Systems by Leslie Lamport
You ask for:

Specification
You ask for:

Specification

You get:

Implementation
Is every behavior of the implementation also a behavior of the specification?
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.q
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)
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
Refinement mapping

1. Map the state of the implementation to the state of the specification
   • Using some function of your choice

2. Show that the initial states of the implementation are also initial states of the specification
   • Using the function above

3. Show that each step of the implementation corresponds to either
   • a state changing step of the specification
   • leaves the specification state unchanged (stuttering step of the spec)
It’s not always possible to get a refinement 😞
Binary Consensus, Specification

Only 0 proposed

0 and 1 proposed

Only 1 proposed

0 chosen

1 chosen

0 learned

1 learned
Paxos

• Value is chosen if a majority 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

• But specification does not allow decided state to revert
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 $S_1$ implements $S_2$ then, possibly by adding history and prophecy variables, there exists a refinement mapping from $S_2$ to $S_1$ (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