Lecture 2
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

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

 TypeInvariant \triangleq isPrime(p)

 Init \triangleq p = 2
 Next \triangleq p' > p \land isPrime(p') \land \forall q \in (p + 1) \ldots (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 ∧ ∀ r in 2..(q-1): q%r /= 0

TypeInvariant == isPrime(p)

Init == p = 2
Next == p' > p ∧ isPrime(p') ∧ ∀ q in (p+1)..(p'-1): ~isPrime(q)

Spec == Init ∧ [] [Next]_p

THEOREM Spec => []TypeInvariant
Some more terms

• A *state function* is a first-order logic expression
• 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 $\sigma$
- 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
  - note that a state predicate is simply an action without primed variables
- If $A$ is an action, then $\sigma \models [A]_\nu$ means that the first step in $\sigma$ is an $A$ step or a stuttering step with respect to $\nu$
Always

• $\sigma \models \Box F$ means that $F$ holds over every suffix of $\sigma$
• More formally
  • Let $\sigma^{+n}$ be $\sigma$ with the first $n$ states removed
  • Then $\sigma \models \Box F \iff \forall n \in \mathbb{N}: \sigma^{+n} \models F$
Boolean combinations of temporal formulas

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

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

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

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

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

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

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* $\sigma \models \Box((x = 1) \Rightarrow \Box(x = 1))$.
Not every temporal formula is a TLA+ formula

• TLA+ formulas are temporal formulas that are \( \text { invariant under stuttering} \)
  • 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!
HourClock revisited

Module HourClock

- **Variable** $hr$
  
  $hr$ is a parameter of the specification HourClock

- $HC_{\text{ini}} \triangleq hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$

- $HC_{\text{nxt}} \triangleq hr' = hr \mod 12 + 1$

- $HC \triangleq HC_{\text{ini}} \land \Box[HC_{\text{nxt}}]_{hr}$
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...

\[ \Diamond \langle A \rangle_v \supseteq \neg \Box [\neg A]_v \]

\[
\sigma \models \Diamond \langle A \rangle_v
\]

\[
\equiv \sigma \models \neg \Box [\neg A]_v \\
\equiv \neg (\sigma \models \Box [\neg A]_v) \\
\equiv \neg (\forall n \in \mathbb{N}: \sigma^+n \models [\neg A]_v) \\
\equiv \neg (\forall n \in \mathbb{N}: \sigma^+n \models (\neg A \lor v' = v)) \\
\equiv \exists n \in \mathbb{N}: \sigma^+n \models (A \land v' \neq v)
\]
HourClock with liveness

clock that never stops

Module HourClock

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

Constant \textit{Data} \hspace{1cm} \textit{Variable chan}

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

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

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

\hspace{1cm} [\text{val} \mapsto d, \text{rdy} \mapsto 1 - \textit{chan}.rdy, \text{ack} \mapsto \textit{chan}.ack ]

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

\hspace{1cm} [\text{val} \mapsto \textit{chan}.val, \text{rdy} \mapsto \textit{chan}.rdy, \text{ack} \mapsto 1 - \textit{chan}.ack ]

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

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

\textit{LiveSpec} \triangleq \textit{Spec} \land \Box(\Diamond(\lnot\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.val \( \in \) Data \& chan.rdy \( \in \) \{0, 1\} \& chan.ack = chan.rdy

Send(d) \( \triangleq \) chan.rdy = chan.ack \& chan' =  
\[ \{ val \mapsto d, rdy \mapsto 1 - chan. rdy, ack \mapsto chan. ack \} \]

Recv \( \triangleq \) chan.rdy \( \neq \) chan.ack \& 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 \& \( \Box \)\( [Next]_{chan} \)

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

Constant Data Variable chan

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

LiveSpec ≜ Spec ∧ □(chan.rdy ≠ chan.ack ⇒ ◊⟨Recv⟩chan)
Weak Fairness as a liveness condition

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

- HourClock: \( WF_{hr}(HCnxt) \)
- Channel: \( WF_{hr}(Recv) \)
(surprising) Weak Fairness equivalence

- $WF_v(A) \equiv \Box(\Box \text{ENABLED} \langle A \rangle_v \Rightarrow \Diamond \langle A \rangle_v)$
  - $\equiv \Box \Diamond (\neg \text{ENABLED} \langle A \rangle_v) \lor \Box \Diamond \langle A \rangle_v$
  - $\equiv \Diamond \Box (\text{ENABLED} \langle A \rangle_v) \Rightarrow \Box \Diamond \langle A \rangle_v$

- Always, if $A$ is enabled forever, then an $A$ step eventually occurs
- If $A$ is eventually enabled forever then infinitely many $A$ steps occur
- $A$ if infinitely often disabled or 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 \textit{continually} enabled
\( WF_v(A) \): an A step must occur if A is \textit{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

Sender

buffer

channels

environment

Receiver
Module Channel

**Constant Data**

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

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

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

**Recv** $\triangleq$ \( \text{chan}.\text{rdy} \neq \text{chan}.\text{ack} \land \text{chan}' = \\
[ \text{val} \mapsto \text{chan}.\text{val}, \text{rdy} \mapsto \text{chan}.\text{rdy}, \text{ack} \mapsto 1 - \text{chan}.\text{ack} ] \)

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

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

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

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

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

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

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

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

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

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

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

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

Receive message from channel \(\text{out}\).
\[ \text{Next} \triangleq \forall \exists \text{msg} \in \text{Message} : \text{SSend}(\text{msg}) \]
\[ \lor \text{BufRcv} \]
\[ \lor \text{BufSend} \]
\[ \lor \text{RRcv} \]

\[ \text{Spec} \triangleq \text{Init} \land \Box[N\text{ext}]_{(\text{in, out, q})} \]

\text{THEOREM} \quad \text{Spec} \Rightarrow \Box \text{TypeInvariant}
Parametrized Instantiation
(not parameterized instantiation 😊)

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

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

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

\[\text{Chan}(\text{in})!\text{TypeInvariant} \equiv \text{in } \in \left[\text{val: Message, rdy: } \{0,1\}, \text{ack: } \{0,1\}\right]\]
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 instance InnerFIFO
spec \triangleq \exists q : inner(q)!spec
Hiding Internal Variables

MODULE FIFO

CONSTANT Message
VARIABLES in, out

Inner(q) ≜ INSTANCE InnerFIFO
Spec ≜ ∃ 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, \( \square \text{len}(q) \leq N \) for some constant \( N > 0 \)
- The only place where \( q \) is extended is in \( BufRcv \)

\[
BufRcv \overset{\Delta}{=} \quad \land \quad \text{InChan}!Rcv
\land \quad q' = \text{Append}(q, \text{in}.val)
\land \quad \text{UNCHANGED} \quad \text{out}
\]
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 \( \text{BufRcv} \)

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

**MODULE BoundedFIFO**

EXTENDS Naturals, Sequences

VARIABLES \( in, out \)

CONSTANT Message, \( N \)

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

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

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

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

If it is a BufRcv step, then \( \text{len}(q) < N \)
**Even cooler (but tricky)**

```plaintext
MODULE BoundedFIFO

EXTENDS Naturals, Sequences
VARiABLES in, out
CONSTANT Message, N
ASSUME (N ∈ Nat) ∧ (N > 0)
Inner(q) ∆ INSTANCE InnerFIFO
BNext(q) ∆ ∧ Inner(q)!Next
∧ Inner(q)!BufRecv ⇒ (Len(q) < N)
Spec ∆ ∃ q : Inner(q)!Init ∧ □[BNext(q)]_{in, out, q}
```